Tutorial Worksheet on OOP with Python

Year One Computing provided an introduction to basic programming in Python but did not cover the object-oriented aspects of the Python programming language in any detail. Instead it implicitly took a *procedural* approach to programming. In procedural programming, there is a clear distinction or separation between variables and data structures on the one hand and the procedures (functions, subroutines, blocks of code) that operate on the data on the other. The programming task is broken down into data structures and procedures that perform operations on the data structures.

In *object-oriented* programming, variables and data structures are bundled together with procedures into *objects*. An object therefore contains both data structures and methods to operate on those data. The object-oriented approach to programming allows a complex programming task to be broken down into self-contained components that can be used by other parts of the program.

The Python programming language fully supports the object-oriented programming paradigm. Although it is possible to ignore the object-oriented aspects of Python and treat it as a procedural programming language, everything in Python is in fact an object. Data structures and variables are objects. Data types are objects. Imported modules are objects. In Python, even functions are actually objects.

Preliminaries

Task 1: Start up Spyder if you have not already done so and create a new project (Menu: Project->New Project…). You may be asked if you want to configure the current workspace: answer yes. Create a new workspace directory on your OneDrive (or "H:" drive) for this and give it a name such as `y2python`. Then create a new project and give it a name such as `y2worksheets`. This will help you keep your program files organised.

NOTE: Although windows allows you to use white-space in file and directory names we strongly recommend you do not. Spaces are often used as the standard token delimiter in terminal applications (including Python). If your module names, scripts or directories contain spaces, you may find it more difficult to use them. For module names we suggest you stick to small-case letters (although if you really must, you could use capitalization such as `myUnnecessarilyComplicatedModuleName.py` to indicate wording.

- Ensure that you have the IPython Console pane visible by checking:

  View -> Panes -> IPython Console.

During these lab sessions you will be expected to run your code interactively in the IPython session. You will be expected to put function definitions, class definitions, etc., inside modules (i.e. **<file>.py** files containing python code), which you can import.

IPython Console tips

This is the 'Interactive Python' console, which you used in Year 1. It is enhanced with **magic commands** prefixed by a % character, that are not part of the Python language, but which you will find useful.

- **`%reset -f`** clears all your variables/objects/functions/modules, so you can start afresh:

```
In [1]: %reset -f
```

- **`%run <script>`** runs the python code in the file `script.py`. More details below.

- **`%time <statement>`** times how long it takes to execute the python statement or expression. (See also `%timeit`.):

```
In [2]: %time sum(range(1000000))
```

- **`%magic`** prints the (rather long) documentation on IPython magic commands to the console.

- **`%lsmagic`** lists the magic commands.

You can get help on a magic command by appending with a **?**:

```
In [2]: %time?
Out[2]:
Docstring:
Time execution of a Python statement or expression.
...
```

Useful Spyder IDE tips

Below are useful `[keyboard shortcuts]` for conveniently running, in the console, portions of code appearing in the editor pane (without copying and pasting). These can also achieved by buttons in the toolbar. Some of these work on **code cells** in the editor. These are denote by lines starting with *#%%*. This was covered last year. (Note that this is *not* part of the Python language.)

```
#%%  --  Start a new code cell  --
l = [1,2,3]
l.append(4)

#%%  --  Start a new code cell  --
a = 2
b = 'hello'
```

- `[F9]` – runs the current line in the editor and advances to the next. If multiple lines are selected, it runs them all in the console.
- `[Ctrl+Enter]` – executes the current cell, i.e., where the cursor is in the editor.
- `[Shift+Enter]` – executes the current cell and advances to the next.
- `[Ctrl+.]` – does a **kernel restart**. This is more drastic than `%reset`. (The python kernel can also be restart from the *cog wheel* button in the console pane upper-right corner).

Here is a convenient PDF crib sheet of all Windows keyboard shortcuts that is freely available on the web.

Running scripts

You can run scripts from the IPython terminal using either of the following:

```
In [4]: %run -i scriptname
```

- This command runs the code in contained in *scriptname.py* as if you had typed it all in at the terminal. (The code executes in the current namespace of the terminal session.) Variables and functions that were defined in the session before the script is run are available to the script, and when the script ends, all variables and definitions made in the script are kept available for subsequent investigation.

```
In [5]: %run scriptname
```

- This second form without the `-i` switch is similar, but rather than running the script in the current session, a fresh namespace is used. At the end of the script, the names defined in the script are still made available to the session for further work as with the first form.

The ability to execute code interactively and to investigate the resulting objects at the terminal afterwards is an important and powerful feature of the interactive interpreter.

Python Essentials

This sections covers some essential material (not covered in year 1 computing) required before you begin to learn object-oriented programming.

Getting help

Python has an intuitive in-built help system. To access it, one uses the help() function. It can be used either with or without arguments as in the examples below:

```
In [6]: help()
```

- Here we call help with no arguments which will bring up the interactive help console. From here we can ask for help about any type/object/module (more later) that Python knows about.

```
In [7]: help(int)
```

- Here we explicitly ask for help about the integer type *int*.

With the IPython extension to Python (which is what we are using), there is yet another way to get help. One can simply append a *?* character on to the type/object/module that is of interest

```
In [8]: int?
```

Introspection

Introspection is the ability to determine information about an object at run time. There are a few Python built-in functions that help us, along with a standard library module called `inspect` (which is beyond the scope of this course).

1. The `dir()` function.

   If called with no arguments it returns the names in the current scope (more later). However it is most often used with an object as an argument. In this case it will return a list of the object's attributes:

   ```
   In [9]: import numpy
   In [10]: dir(numpy)
   Out[10]: [..., 'arccosh', 'arcsin', ..., 'array', ...]
   ```

2. The `type()` function.

   This function returns the type of an object:

   ```
   In [11]: type(12)
   Out[11]: <class 'int'>
   ```

3. The `id()` function.

   This function returns a unique id for a given object:

   ```
   In [12]: id(numpy.array([1, 2, 3]))
   Out[12]: 140299842849968
   ```

   - The id returned will be different on your machine but will be unique for each object.

String Formatting

There are several ways to format strings in python we will only discuss the most basic as part of this course.:

- Old style uses printf-style (C-style) formatting. With this method, a formatting/interpolation operator *%* marks the locations in the string where we wish to make a substitution. There usually follows a conversion flag that denotes the type to convert the substitution variable into, in the following example *s* means string and *d* means integer. In order to specify the variable that we want to insert we add the *%* character immediately after the string followed by the variable's name:

  ```
  In [13]: name = 'bob'
  In [14]: "Hello World, my name is %s" % name
  Out[14]: 'Hello World, my name is bob'
  ```

  In the above example we substituted a single variable but we can in general substitute as many as we like. The only modification required is that the variables to substitute are given in a tuple as follows:

  ```
  In [15]: name = 'bob'
  In [16]: age = 35
  In [17]: "Hello World, my name is %s and I am %d years old" % (name, age)
  Out[17]: 'Hello World, my name is bob and I am 35 years old'
  ```

  We can also use a dictionary of values to substitute based on their key name as follows:

```
In [18]: substitutes = {'name': 'bob', 'age': 35}
In [19]: "Hello World, my name is %(name)s and I am %(age)d years old" % substitutes
Out[19]: 'Hello World, my name is bob and I am 35 years old'
```

Full information about this style of string formatting (along with a complete list of conversion flags e.g. the *s* and *d* after the formatting/interpolation operator *%*) can be found in the Python 3 string formatting documentation.

- New style formatting is more versatile than the basic method outlined above but we won't have time to cover it here. If you are interested you can read the Python string formatting documentation.

- Literal string interpolation is new to Python 3.6 and again not covered by this course but if your interested then you can see PEP 498.

Lists

While you may have used lists extensively last year, you may not have realised that they are a very general form of *container*.

- Their elements do not have to be of the same type.
- They can contain built-in data types, objects, classes, modules and even functions.
- Lists can even contain other lists.

```python
import numpy
import random

def my_func(i, j, k):
    return i, j, k

a = [1, 2, 3]
b = [1, numpy.array([2, 3, 4]), 5]
c = [random, my_func, numpy.ndarray]
d = [1, [2, 3, 4], 5]
```

You can conveniently check if an object exists in a list using the `in` keyword as below:

```
>>> 5 in d
True
>>> [2, 3, 4] in d
True
>>> ('hello' in d, 1 in d)
(False, True)
```

Equality Vs Identity

Assignment and equality a two very different notions in programming. In Python they correspond to using the = or == operators. The similarity of these operators is a common stumbling block for beginners and I'm sure you remember these from last year's course. Less well understood however is the difference between equality == and identity which in Python can be determined using the `id()` function or the **is** keyword.

Task 2: We will investigate the equality and identity of some variables. By typing directly into your IPython console session, Consider the following:

```
a = 100
b = a
```

What is the result of the following?:

```
a == b
a is b
```

At first glance it would appear that these two statements are synonymous, however what happens if you try the same thing with these variables?:

```
c = [1, 2, 3]
d = [1, 2, 3]
```

Equality is checking if the content of the two variables are equal in terms of their value. The identity check however is asking if the two variables point to the **same** object in memory.

- In the example above although the two list have the same values in them they are different objects and hence the identity check fails.

Task 3: Given the results of the task above what do you expect to be the output from checking the equality and identity of the following?:

```
e = [1, 2, 3]
f = e
```

Now try the modifying the content of *e* and *f* as follows:

```
e[0] = 100
f[1] = 200
```

- What do you notice about the values of *e* and *f* now?
- Does this make sense given what you know about identity from above?

Copying

The operation in the task above is **not** copying, we are merely setting both variables *e* and *f* to point at the same list object in memory hence changes to one will affect the other.

Task 4: So now let’s actually make a copy. Sticking with the above example we can do that as follows:

```
g = [1, 2, 3]
h = g.copy()
```

Again try the following:

```
g[0] = 100
h[1] = 200
```

- What do you see this time when you print the values of *g* and *h*?

- Prove, using the **is** keyword that *g* and *h* refer to distinct objects.

So we have created a new object which is a copy of the first…

However this is not the end of the story.

Task 5: Try the following:

```
i = [1, 2, 3, [1, 2, 3]]
j = i.copy()

i[0] = 100
j[1] = 200
i[3].append(4)
j[3].append(5)
```

- What do you notice about the values of *i* and *j*?
- What do you think is going on here (as we know from the task above that *i* and *j* should be different objects)?

Task 6: The key to understanding what's going on is to try the following:

```
i[3] is j[3]
```

- What is the result of the above and what does it mean?

This is an example of a **shallow** copy. In a shallow copy, a new object is created with the same values at the original except that any objects that are contained within are merely pointed to.

A **deep** copy by comparison would not only create a duplicate object but also new copies of all the objects to which it refers. In this example it can simply be achieved using the `deepcopy()` function.

Task 7: Try the following:

```
from copy import deepcopy
k = [1, 2, 3, [1, 2, 3]]
l = deepcopy(k)
```

Now try the familiar modifications:

```
k[0] = 100
l[1] = 200
k[3].append(4)
l[3].append(5)
```

- Are the values of *k* and *l* now independent?
- What is the result of the following identity check?:

  ```
  k[3] is l[3]
  ```

Modules

A module is a file containing Python code. The name of the file is of the form `modulename.py` where `modulename` is the name of the module. When you import a module, the code in the file is loaded and put into a module namespace called `modulename`. You have already used existing library modules in Year One Computing.

Task 8:

- Ensure that the Project Explorer pane is visible by checking the following:

  View -> Panes -> Project Explorer

Now create a new Python module called `mymodule.py` in your newly created project (right-click on your project in the “Project explorer” pane and choose New->Module from the context menu.)

If it hasn’t already, double-clicking `mymodule.py` in the project explorer pane will open it in an editor pane.

Using the editor, type just the following text in it:

```
"""
An example module
My Name
"""
```

Save the file.

- The IPython console session should have the working directory set to the project directory where `mymodule.py` resides. If not, you should set this now. (You can change directory in the IPython terminal using `cd` and you can print the current directory using `pwd`, or you can set the working directory for new sessions near in the text-box located near the top-right of the Spyder window.)

Import the module:

```
>>> import mymodule
```

What names are defined in the module? Try **`help()`** on the module.

Task 9: Now using the text-editor, add the following text to the module file (you should be able to cut-and-paste):

```
"""
An example module
C Paterson  30/9/2013
"""

import numpy as _np

_sqrt5 = _np.sqrt(5.0)
_phi = (1.0+_np.sqrt(5.0)) / 2.0
```

```
def fibo1(n):
    "Implement method 1 to calculate a Fibonacci number"
    val= int(_phi**n/_sqrt5 + 0.5)
    return val

def fibo2(n):
    "Implement method 2 to calculate a Fibonacci number"
    val= (_phi**n - (-_phi)**(-n)) / _sqrt5
    return val

def fibo(n):
    "Calculate the nth Fibonacci number"
    return __fibo_impl(n)

__fibo_impl = fibo1

__all__ = ['fibo']
```

(Or you can get a copy of the modified file here : mymodule.py)

Now what extra names are defined in the module? Is the result what you would expect?

When you try to import a module `mymodule` using the **`import`** directive, Python searches the import paths (starting with the current directory, before moving on to system library paths) for a module file with the name `mymodule`. If Python finds such a file, it imports the contents into a module with the name `mymodule`. The module file can contain definitions and executable statements to initialise the module that are executed when the module is imported.

However, if a module with that name already exists in one of Python's namespaces (in other words, the module has already been imported), then the file is not executed again if you issue a second:

```
import mymodule
```

This means if you made changes to `mymodule` then simply importing it again would not reflect these changes. So why did your changes above appear without an import…?

When you make modifications to your modules and save them within the Spyder IDE however, your modules that have already been imported will be automatically be reloaded. This behaviour is new and used to require a manual reload. This is why your changes to `mymodule` in the above task took immediate effect.

Note: This is not standard python behaviour, rather a convenience provided by Spyder. The standard way to reload a module after it's been imported involves using the **`reload()`** function. An example is given in the Reloading Appendix.

Note: Objects that you create from your classes (to be introduced later) will not automatically reload changes even if the class they are based on does. This means you will have to create the objects fresh each time.

Note: If you get in a pickle after modifying your module and saving it, performing a `%reset` (or failing that, a kernel restart) and running your code again will fix things.

Task 10: Now try to use each of the functions. To access the function you must prepend the module name with the dot notation. e.g., `mymodule.func(50)`. Note that code defined inside the module can access names defined in the module directly since they are in the same namespace. (More on this later.)

In fact, there are alternative ways of using the **`import`** directive:

```
>>> import mymodule as mym
>>> mym.fibo(10)
>>> from mymodule import fibo1
>>> fibo1(20)
>>> from mymodule import *
```

The last two of these import the specified names directly into the current namespace (more to come). The `*` form imports all available names into the namespace.

If the module defines **`__all__`**, it is used to determine which names get imported using the `from mymodule import *` form. **`__all__`** must be a list of strings containing the names to be imported. This is a convenient way of limiting the names that get imported only to those that the user of the module might need. (Even if **`__all__`** is not defined, names beginning with an underscore do not get imported into the current namespace.)

Try editing the mymodules.py file, and check you understand this behaviour.

Commenting

Commenting is a common practise in all languages. A comment is a line of text which is not deemed to be code but is placed there for the benefit of the programmer or indeed anyone reading the code. As such it is ignored by the Python interpreter.

Commenting can be useful to add a small note to a rather complicated chunk of code so that upon revisiting it can be quickly understood.

Note: Comments are **NO** substitute for good documentation strings. Doc strings are readable by the users of your code and provide valuable information about how it works or how to use it. They serve as a built in manual.

Comments in Python start from the # character and run to the end of the line. Note that it is **not** necessary for the comment starting character to begin a line, so you are free to start a comment after some valid Python code:

```
"""
Module docstring
"""
# This is a comment

a = 12  # The rest of this line is a comment

def func():  # The rest of this line is a comment
    """
    Function doc string
    """
    return 12
```

Unlike other languages, Python does not provide any syntax for a block comment. This means that long comments need to have a # on each line, essentially numerous single line comments:

```
# This is a long comment
# that spans multiple lines.
```

This is clearly annoying and to circumvent this many people will use a block string literal instead. You have seen these already as docstrings to functions/modules:

```
"""
This is a multiline
string literal
"""
```

Note:

- These are technically not comments but since they are not bound to a variable they will not cause problems.
- It is not recommended to do this for comments in finished code however. Far better to just write smaller more concise proper comments. Where this can be useful is if (for debugging purposes) you need to comment out large chunks of code.

Namespaces and scope

A namespace is a mapping from names to objects. At different points during the execution of a program there will be different namespaces available depending on the current scope. A scope is a section in the code over which a namespace is accessible, for example if the current point of execution is inside a function `func()`, then the scope is the section of code that makes up that function. At any given point during the execution there will be several namespaces present depending on the nested scopes:

- the innermost namespace, or the local namespace — when inside the execution of a function, the function's namespace is the innermost namespace,
- the module namespace — this contains the current module's global names, (by global we mean those not defined inside other functions or inside other objects in the module),
- the outermost namespace — this contains the built-in functions.

Namespaces are created at different points during the execution.

- When the IPython console is started (or when a script is run) a pseudo-module called `__main__` is created. Variables defined in the console or the run script go into it's namespace.
- When a module is imported, all the variables/functions/classes that it defines are grouped into a namespace which is accessible by using the name of the module as a prefix, e.g. *modulename.variable* or *modulename.function*.
- A function's local namespace is created whenever the function is entered, and it is forgotten when the function returns. Each invocation of the function gets its own namespace.

See the following example:

```python
"""
Some Module.
"""

print("Loading my module.")
# This print function is not defined here but exists within the 'built-in' namespace.
# we always have access to the 'built-in' namespace.

my_global_var = 123
# At module scope we have access to only the 'global' and 'built-in' namespace.
# This namespace is created when we first load our module. Any variables that we
# define here (like my_global_var) are called global variables.

def funcA(a, b, c):
    "Some function."
    my_local_var = a * b + c
    # Inside this function scope we have access to the 'local', 'global' and 'built-in'
namespaces.
    # This namespace is recreated each time we enter the function and contains amongst
    # other things the args a, b and c as well as any variables we define here
    # (like my_local_var) which are hence called 'local' variables.
```

During execution of the code, the interpreter searches for names in the various namespaces. Unless a name is declared to be global, the namespaces are searched starting with the innermost (corresponding to the local scope) until the name is found. Names in the innermost namespace can be assigned to, but names found in the other namespaces are read-only. Any attempt to assign to a name causes a new entry to be created in the local namespace if it does not already exist there, even if the name already exists in one of the outer scopes. When a function is parsed, if there is any assignment to a name in the function, that name is assumed to be local in the function.

Task 11: Create a module called coffeecake.py with the following content:

```python
"""
A module to investigate namespaces and scope.
C Paterson 30/9/2013
"""

a = "coffee"
b = "cake"

def func1():
    a = "drunk"
    b = "eaten"
    return a, b

def func2(a, b):
    a = "drunk"
    b = "eaten"
    return a, b

def func3():
    c = a
    return c

def func4():
    c = a
    return c
    a = c

def func5():
    global a
    a = "drunk"
    b = "eaten"
    return a, b
```

You can get the file here: coffeecake.py (You may need to right-click to save the file.)

Import the module in a new IPython terminal session with the following:

```
>>> import coffeecake as coca
```

Then try each of the following at the prompt (you can just cut and paste to the terminal to save typing). Read through the function definitions above first and try to predict what each test will produce:

```
coca.func1()

coca.a, coca.b
```

```
coca.func2(coca.a, coca.b)

coca.a, coca.b
```

```
coca.func3()
```

```
coca.func4()
```

```
coca.func5()

coca.a, coca.b
```

Can you see why `func4()` raises an error, but `func3()` does not?

This behaviour can be overridden: if a name is declared `global` then the name in the module scope is used as if it were local.

Task 12: The `locals()` and `globals()` functions return dictionaries of the local and global namespaces at their point of invocation.

Add the following functions to coffeecake.py:

```
def func6():
    a = "drunk"
    return locals()

def func7():
    a = "empty"
    return globals()
```

Call these functions from the IPython session. Do they return what you expect?

Stop 1: Before starting to learn OOP, check your work and understanding with a demonstrator. Ask them at this point (if you haven't already) if there is anything that is not clear from the above content.

OOP1: Classes and Objects

The class is a template for making objects. The class construct is how one defines a new type of object. In Python, we define a new class using the keyword **`class`** in the following way:

```
class ClassName:
     <statement>
     <statement>
     ...
```

So for example we can define a new class Counter:

```
class Counter:
    def __init__(self):
        self.i = 0
    def addone(self):
        self.i += 1
    def reset(self, i=0):
        self.i = i
```

Then we can create objects of type `Counter` using *instantiation*. In Python we use function notation to *instantiate* an object. So to create a new `Counter` object we use the following:

```
>>> c = Counter()
```

This creates a new *instance* of the class and assigns it to the name **`c`**. If the class has a method called **`__init__()`** it gets called during instantiation and can be used to initialise the object. In this case to create the attribute **`i`**.

We can access the attributes of **`c`** using the dot notation:

```
>>> c.i
0
>>> c.addone()
>>> c.addone()
>>> c.i
2
>>> c.reset(5)
>>> c.i
5
```

A class definition creates a new scope, the class scope. Names declared in the class are available inside the class scope.

A newly created *instance* object has two types of attributes: data attributes and methods. Figure 1 shows a schematic of a general class (left) and the complex number class (right) that will be developed as an example below.

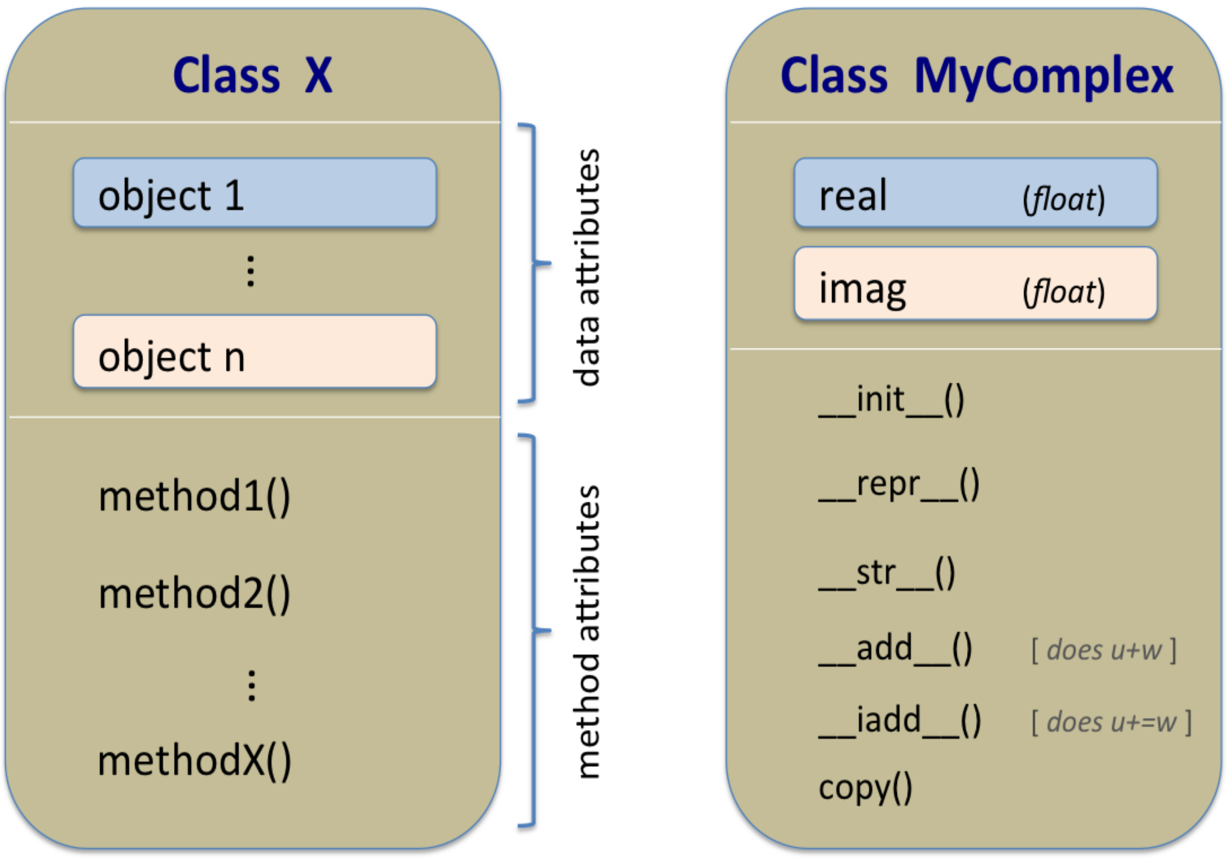


Fig 1. Schematics of classes, built out of exisiting objects. (Left) General idea. (Right) Complex number example in the script. This is known as *composition*.

Data attributes

Data attributes are data that are held by the instance. In this case, `c.i` is a data attribute of `c`. Each instance of the class `Counter` gets its own set of data attributes:

```
>>> c1 = Counter()
>>> c2 = Counter()
>>> c2.addone()
>>> c2.addone()
>>> c2.i
2
>>> c1.i
0
```

Method attributes

The other type of attribute is a *method*. *Methods* are functions that belong to the object. In this case **`c.addone()`** and **`c.reset()`** are method attributes. Any function that is defined in the class gives rise to a corresponding method attribute in the instance. Note however that a method is not the same as the corresponding function defined in the class.

A method attribute can be thought of as a reference to the function in the class. When one calls the method, the class function is called with the instance inserted into the parameter list as the first parameter. Therefore, `c.addone()` is equivalent to `Counter.addone(c)` This allows **`addone()`** to have access to the attributes of the instance object. Similarly, `c.reset(5)` is equivalent to `Counter.reset(c, 5)`.

Notice that the first argument in the definitions of the methods of the class is called `self`. In fact this is only a convention: there is nothing special about the name `self` and you could use whatever you like. However, it is a **VERY strong convention** and if you want your code to be readable to other Python programmers you are strongly encouraged to stick to it. (Similarly, it is a good idea not to use the name `self` for anything other than the first argument in a class method function.)

Initialization: `__init__()`

It is often useful to be able to perform initialization on the newly created instance object. If the class contains a function `__init__()`, then this method is called automatically during the instantiation Consider the following example class to implement a complex number (Note: in practice we would not actually write our own complex number class - Python already has perfectly good complex numbers built in!)

```
class MyComplex:
    real = 0.0
    imag = 0.0
```

```
>>> c = MyComplex()
>>> c.real = 42.0
>>> c.imag = 24.0
>>> c
<MyComplex.MyComplex at 0x10df21b70>
>>> c.real,  c.imag
(42.0, 24.0)
```

We can instead add a function `__init__()` to the class to perform initialization, thus:

```
class MyComplex:
    def __init__(self, re=0.0, im=0.0):
        self.real = re
        self.imag = im
```

Note that we have provided default values for the parameters so we can instantiate objects in a variety of ways:

```
>>> c1 = MyComplex(42.0, 24.0)
>>> c1.real, c1.imag
(42.0, 24.0)
>>> c2 = MyComplex(42.0)
>>> c2.real, c2.imag
(42.0, 0.0)
>>> c3 = MyComplex()
>>> c3.real, c3.imag
(0.0, 0.0)
>>> c4 = MyComplex(im=24.0)
>>> c4.real, c4.imag
(0.0, 24.0)
```

Note also that we have no longer defined `real` and `imag` in the body of MyComplex. In Python we can add data attributes to an object at any point, in this case inside the __init__ function.

Data attributes spring into existence as soon as we define them and do not need to be pre-declared. Our new class `MyComplex` no longer has the **`real`** and **`imag`** attributes, but they are created by the assignments in **`__init__()`**.

```
>>> dir(MyComplex)
['__doc__', '__init__', '__module__']
>>> c = MyComplex()
>>> dir(c)
['__doc__', '__init__', '__module__', 'imag', 'real']
```

In fact we can add attributes at any time to an instance:

```
>>> dir(c)
['__doc__', '__init__', '__module__', 'imag', 'real']
>>> c.comment = "I'm a confused complex number!"
>>> dir(c)
['__doc__', '__init__', '__module__', 'comment', 'imag', 'real']
>>> c.real, c.imag, c.comment
(0.0, 0.0, "I'm a confused complex number!")
```

Docstrings in classes

Note that our class also has a couple of other attributes that we have not explicitly defined (**`__doc__`** and **`__module__`**.) **`__module__`** is the name of the module in which the class was defined. **`__doc__`** can be used for documenting the class. We can define **`__doc__`** explicitly if we wish:

```
class MyComplex:
    def __init__(self, re=0.0, im=0.0):
        self.real = re
        self.imag = im

    __doc__ = """
    Another complex number class
    (WARNING: you probably want to use Python's built-in complex instead)
    """
```

However, it is more usual (and more readable) to use a docstring immediately inside the class definition block:

```
class MyComplex:
    """
    Another complex number class
    (WARNING: you probably want to use Python's built-in complex instead)
    """

    def __init__(self, re=0.0, im=0.0):
        self.real = re
        self.imag = im
```

If we use help on our class (or indeed on any instances of the class), our docstring will be used:

```
>>> help(MyComplex)
Help on class MyComplex in module __main__:

class MyComplex
 |  Another complex number class
 |  (WARNING: you probably want to use Python's built-in complex instead)
 |
 |  Methods defined here:
 |
 |  __init__(self, re=0.0, im=0.0)
lines 1-9/9 (END)
```

In relativity a 4-vector is a vector in a four-dimensional space (Minkowski space.) You should already have come across 4-vectors in the year one Relativity lecture course. If not, or you need a refresher see the 4-Vectors Appendix. A 4-vector is a vector with three space-like components and one time-like component.

Task 13: Write a class `FourVector`. You should be able to initialize instances with space-like and time-like parameters. You will need to store the four components inside the object, in data attributes. For this example, *store* the three space-like components as a NumPy array in a with name *r* and the time-like component as a number in a with name *ct*.

Make sure that your `FourVector` class allows the following ways of instantiation all to work for your FourVector:

```
P0 = FourVector()  #should initialize to zero
P2 = FourVector(ct=99.9, r=[1, 2, 3])
```

Notice carefully that the *initialization* above is achieved with parameter *r* specified as a `list` rather than a NumPy array. This is desirable for both convenience and other reasons.

[Hint: see Year One, Session 3, section 4. Functions (*Optional Keywords*) for an example of how to call functions with optional arguments using default values.]

Give your class a docstring.

Put your class inside a new module called `relativity`.

Check that you can correctly instantiate `FourVector` objects and that your instances are independent of each other.

`__repr__()` and `__str__()`

When we type the name of an object at the IPython terminal, a representation of the object is printed as a string, e.g.,:

```
>>> r = numpy.arange(4)
>>> d = dict(name='Ignavus', feedback='Work harder.')
>>> r
array([0, 1, 2, 3])
>>> d
{'feedback': 'Work harder.', 'name': 'Ignavus'}
```

If we do this for our MyComplex class we get the following:

```
>>> c = MyComplex(3, 4)
>>> c
<mycomplex.MyComplex instance at 0x1ea56c8>
```

Although this tells us something about object `c` (that we have a MyComplex instance at a particular memory location), we would probably be more interested in seeing the real and imaginary components. Fortunately, in Python we can determine how our objects are represented as strings.

When we type a name at the command line, what actually happens is that the *built-in* function **`repr()`** is called to produce a string representation of the object. We could call **`repr()`** on any object directly if we wish:

```
>>> repr(d)
"{'feedback': 'Work harder.', 'name': 'Ignavus'}"
```

But how does **`repr()`** know how to deal with the different types of objects that it could be called with? One approach might be for **`repr()`** to contain all the code necessary to print every conceivable object and check for the type of each object. It does not take much thought to realise that this approach would become rather inflexible, especially if we wanted to add our own types, such as classes. In fact what happens is that the responsibility of producing its own string representation is *delegated to the object*. **`repr()`** just checks to see if the object has a method called **`__repr__()`** and calls that to get the string. This concept of delegating the responsibility to implement well defined behaviour to the object itself is a key advantage to the object-oriented approach to programming. In this case **`repr()`** does not need to know about the internals of the object. All it needs to know is that the object provides the right method **`__repr__()`**.

To make use of this we can provide our own class with a method function **`__repr__()`** which should return a string representation of the object.

We shall define **`__repr__()`** method for MyComplex class:

```
class MyComplex:
    ...
    def __repr__(self):
        return "%s(re=%g, im=%g)" % ("MyComplex", self.real, self.imag)
```

(If you can't remember how to format strings refer back to the string formatting section at the start of this worksheet and follow the link given for a more detailed description.)

This gives:

```
>>> c = MyComplex(3, 4)
>>> c
MyComplex(re=3, im=4)
```

As well as **`repr()`** we can also use **`str()`** to get a string from the object. **`str()`** calls the object's method **`__str__()`** if it is defined, which should also return a string representation of the object.

So, what is the difference between **`__repr__()`** and **`__str__()`**? Both should return a string representation of the object. However their purposes are slightly different:

- **`__repr__()`** provides the *official* representation of the object: the emphasis should be on a precise and unambiguous representation. It is used more during development and debugging. Ideally the string returned by **`__repr__()`**

should be a valid Python expression that could be used to re-create the object itself. When you type a name at the IPython prompt you get the string returned by `__repr__()`.

- `__str__()` returns the *informal* representation of the object: the emphasis should be on a simple, easy-to-read (human-oriented) representation of the object. `__str__()` is called whenever you use `print()` (or more directly using the `str()` built-in).

Defining both `__repr__()` and `__str__()` methods for MyComplex class:

```
class MyComplex:
    ...
    def __repr__(self):
        return "%s(re=%g, im=%g)" % ("MyComplex", self.real, self.imag)

    def __str__(self):
        return "%g + %gi" % (self.real, self.imag)
```

we then get:

```
>>> c = MyComplex(3, 4)
>>> c
MyComplex(re=3, im=4)
>>> print(c)
3 + 4i
```

Note that the string returned when we type `c` at the prompt is valid Python (we could cut and paste it back into the terminal to make a new instance.) When we type `print(c)` we get a simpler, more human-readable representation.

We are not *required* to define these methods if we do not wish to — our class will still function without them. However it is usually a good idea. Of the two, `__repr__()` is the more important as it it can be particularly useful during development and debugging. If we only provide `__repr__()` but do not provide `__str__()`, then `__repr__()` will be used in its place.

Note: Up until now when formatting strings you have likely been using the *%s* conversion flag when you wanted to substitute in a string value. What you probably didn't realise was that this calls `str()` on the variable to be substituted to convert it to a string. This is why you can use *%s* with an integer variable. You can equally call `repr()` on the substitution variable by using the conversion flag *%r*.

Task 14: Write `__repr__()` and `__str__()` methods for your `FourVector` class.

For example:

```
>>> v = FourVector(ct=99, r=[1.0, 2.0, 3.0])
>>> v
FourVector(ct=99, r=array([ 1.,  2.,  3.]))
>>> print(v)
(99, 1, 2, 3)
```

Private attributes and name mangling

When designing a class it is a good idea to make a clear distinction between the interface provided to users of the class and the internal workings of the class. To use the class it should not be necessary to know about the internal workings of the class. In fact if a user (inadvertently) accesses or changes some of the internal parts of an object, this may result in unintended behaviour. It is useful therefore if attributes of the class that are intended for internal use only marked as such and/or hidden from the user. In Python, we can do this giving such attributes names with a leading underscore. If we give an attribute a name of the form `_name` we are indicating that it is not intended for public use: it can be used from within the methods of the class to implement behaviour, but should not be used outside the class. (There is nothing actually stopping a user from accessing `_name` if they want to of course.)

We can make the message stronger with two leading underscores. If we give an attribute a name with two leading underscores, `__name`, the name gets *mangled* and is not directly accessible outside of the class.

```python
class MyClass:

    def __init__(self):
        self.public_variable = 123
        self._hidden_variable = 'abc'
        self.__mangled_variable = 'abc123'

    def some_method(self):
        # all can be accessed from within the class
        print(self.public_variable)
        print(self._hidden_variable)
        print(self.__mangled_variable)
```

Then:

```
>>> C = MyClass()
>>> C.some_method()
123
abc
abc123
>>> C.public_variable
123
>>> C._hidden_variable  # We can actually access these hidden variables but should avoid
doing so
'abc'
>>> C.__mangled_variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyClass' object has no attribute '__mangled_variable'
```

Note that `__mangled_variable` attribute is not available from outside the class. It is still possible for someone who is determined to access it. We can still find `__mangled_variable` if we look for it, but its name has been changed (this is what we mean by *name mangling*):

```
>>> dir(C)
['_MyClass__mangled_variable', ...,  '_hidden_variable', 'public_variable', 'some_method']
>>> C._MyClass__mangled_variable
'abc123'
```

The Python approach is not to prohibit access, but rather to suggest to you that you should not access it. (Python treats its users as “consenting adults”, rather than children who need strict rules.)

Task 15: Modify your `FourVector` class so that the data attributes are *hidden*.

Provide instead methods `ct()` and `r()` to return the values of the time- and space-like components. (In OOP such methods are categorised as *access methods*.)

Also provide methods `setr(new_r)` and `setct(new_ct)` to modify the values of the data attributes. (In OOP such methods are categorised as *modifier methods*.)

Copying

The assignment operator in Python does not create objects, rather it just attaches the name on the left of the `=` to the object on the right:

```
>>> c1 = MyComplex(3, 4)
>>> c1
MyComplex(re=3, im=4)
>>> c2 = c1
>>> c2.real = 0
>>> c1
MyComplex(re=0, im=4)
```

It is often necessary to be able to produce a distinct copy. It is useful to provide a `copy()` method that returns a separate copy of the instance with the same values. For `MyComplex` this is rather trivial:

```
class MyComplex:
    ...
    def copy(self):
        return MyComplex(self.real, self.imag)
```

Then:

```
>>> c1 = MyComplex(3, 4)
>>> c1
MyComplex(re=3, im=4)
>>> c2 = c1.copy()
>>> c2.real = 0
>>> c1
MyComplex(re=3, im=4)
>>> c2
MyComplex(re=0, im=4)
```

Task 16: Write a `copy()` method for `FourVector`. Check that the returned instance has the same values but is independent of the original.

Stop 2: Check your code is sensibly documented and then check your work with a demonstrator.

Arithmetic

If we wanted to add two complex numbers we could do something like the following:

```
>>> c1 = MyComplex(3, 4)
>>> c2 = MyComplex(2, -1)
>>> c3 = MyComplex(c1.real + c2.real, c1.imag + c2.imag)
>>> c3
MyComplex(re=5, im=3)
```

This is rather cumbersome, so it would be useful to implement useful functions such as arithmetic operations as methods. We could do this as follows:

```
class MyComplex:
    ...
    def add(self, other):
        return MyComplex(self.real + other.real, self.imag + other.imag)
```

```
>>> c1 = MyComplex(3, 4)
>>> c2 = MyComplex(2, -1)
>>> c3 = c1.add(c2)
>>> c3
MyComplex(re=5, im=3)
```

However it would be even more natural to be able to write `c3 = c1 + c2`. We can achieve this using special method names. A class can implement operations that can be invoked using such operator syntax by defining methods that have special names. So for example if we want to be able to use the addition operator `+` between two objects, we must provide the method **`__add__()`**. For the operator `+=` addition we provide the method **`__iadd__()`**. Thus for **`MyComplex`** we would write:

```
class MyComplex:
    ...
    def __add__(self, other): # implement +
        return MyComplex(self.real + other.real, self.imag + other.imag)

    def __iadd__(self, other): # implement +=
        self.real += other.real
        self.imag += other.imag
        return self
```

We could then use these methods with operator syntax:

```
>>> c1 = MyComplex(3, 4)
>>> c2 = MyComplex(2, -1)
>>> c3 = MyComplex(10, 10)
>>> c4 = c1 + c2
>>> c4
MyComplex(re=5, im=3)
>>> c1 += c3
>>> c1
MyComplex(re=13, im=14)
```

This is referred to as *operator overloading*. There are many possible special method names that can be provided to implement a wide range of operations (arithmetic, comparison, slices and indexing, etc.) See http://docs.python.org/3/reference/datamodel.html#special-method-names for a detailed description. Which of these it makes sense to implement depends on the object.

Task 17: Write the methods to implement `+`, `+=`, `-`, `-=` for `FourVector`. (For example, arithmetic of energy-momentum four-vectors could be used to represent conservation in a system of particles.)

The inner product between two 4-vectors $\mathbf{v}_1 = (ct_1, \mathbf{r}_1)$ and $\mathbf{v}_2 = (ct_2, \mathbf{r}_2)$ is given by

$$\mathbf{v}_1 . \mathbf{v}_2 = ct_1 ct_2 - \mathbf{r}_1 . \mathbf{r}_2$$

Then the square of a 4-vector is $\mathbf{v}_1 . \mathbf{v}_1$.

Task 18: Write methods `inner()` and `magsquare()` for `FourVector` to implement inner product and the square.

Task 19: Write a method `boost()` that implements a Lorentz boost in the z-direction. Your method should take β as its parameter.

Apply various combinations of Lorentz boosts to some example four-vectors. Verify that the square does not change.

Exceptions

You will certainly already have encountered Python exceptions - whenever something goes wrong in the code and you see something like this:

```
>>> x = 42 + "The question"
Traceback (most recent call last):
  File "<ipython-input-102-3f0638adef2a>", line 1, in <module>
    x = 42 + "The question"
TypeError: unsupported operand type(s) for +: 'int' and 'str'

>>> x = 1 / 0
Traceback (most recent call last):
  File "<ipython-input-1-b710d87c980c>", line 1, in <module>
    x = 1 / 0
ZeroDivisionError: division by zero
```

Exceptions are class objects and there are many different types — in these examples, `TypeError` and `ZeroDivisionError`. For a list of the built-in exceptions and their uses see http://docs.python.org/3/library/exceptions.html#concrete-exceptions .

We can raise exceptions in our own code using the `raise` construct. Try the following example:

```
for t in range(20):
    if t > 5:
        raise Exception("too late")
    print(t)
```

Let's return to the initialization function of the `FourVector` class. You were asked to implement the initialization so that the following example would work:

```
>>> v = FourVector(ct=99, r=[1.0, 2.0, 3.0])
```

However, what happens if you try this?

```
>>> v = FourVector(ct=99, r=[1.0, 2.0, 3.0, 4.0])
```

It is not obvious what this would represent, although it might not cause your code to complain, at least not initially. If later on we tried to use the resulting FourVector in a calculation, we might get an error — or worse: no error, just incorrect results. We could waste a lot of time trying to track down the problem. Instead of waiting for the problems to get worse, we can test for the length of parameter `r` and raise an exception if it is wrong.

Task 20: Modify your **`FourVector`** class so that if it is instantiated with a parameter `r` that is the wrong size, you code raises an appropriate exception.

Test your code: it should produce something like this:

```
>>> x = FourVector(1.0, [1, 2, 3, 4])
Traceback (most recent call last):
  File "<ipython-input-42-854051d2cd67>", line 1, in <module>
    x = FourVector(1.0, [1, 2, 3, 4])
  File "<ipython-input-41-406a3b739f9e>", line 7, in __init__
    raise Exception("FourVector parameter r has incorrect size")
Exception: FourVector parameter r has incorrect size
```

When an exception is raised using `raise myexception`, the program jumps out of the current point of execution to the nearest exception handler, which either deals with the exception, or if it cannot passes it on to the next exception handler etc, ultimately leading to the program exiting if the exception cannot be dealt with. (Writing code to deal with exceptions is not difficult, but it is beyond the scope of this short lab course. For a quick introduction see http://docs.python.org/3/tutorial/errors.html)

Exceptions provide a convenient way of interrupting the current flow of the program - not only in the cases where something has gone wrong. For example, they are an essential part to how Python deals with iterations and for-loops.

Stop 3: Check your work with a demonstrator for feedback.

OOP2: Inheritance

Inheritance is one of the important features of OOP. It allows you to build a new class (called a *derived class*) by extending an existing class (called a *base class*). The derived class automatically contains the methods and attributes of the base class, although these can be overwritten. This generally makes it easier to create and maintain your programs and allows you to utilise the code in a base class in several derived classes. It also means that by changing the code in your base class you change the behaviour of the derived classes in a common way.

(Be aware that other terminology for *base class* exists in OOP: *superclass* and *parent class*. Similarly in OOP, *derived classes* are often also referred to as *subclasses* and *child classes*.)

The syntax to create a derived class from a base class is:

```python
class MyDerivedClass(BaseClass):
```

Consider the following example:

```python
class Animal:
    """ a made up class to act as a base class
    for a teaching example
    DJC- October 2013, Revised AMacK Oct 2016"""

    def breaths(self):
        return True

    def likes_to_eat(self):
        return "food"


class Carnivore(Animal):
    """A derived class which inherits from animal """

    def likes_to_eat(self):
        return "Meat!"

    def has_sharp_teeth(self):
        return True

class Herbivore(Animal):
    """ another derived class which inherits from animal """

    def likes_to_eat(self):
        return "plants"

    def has_flat_teeth(self):
        return True
```

Diagrams for the above example classes are shown in Figure 2.

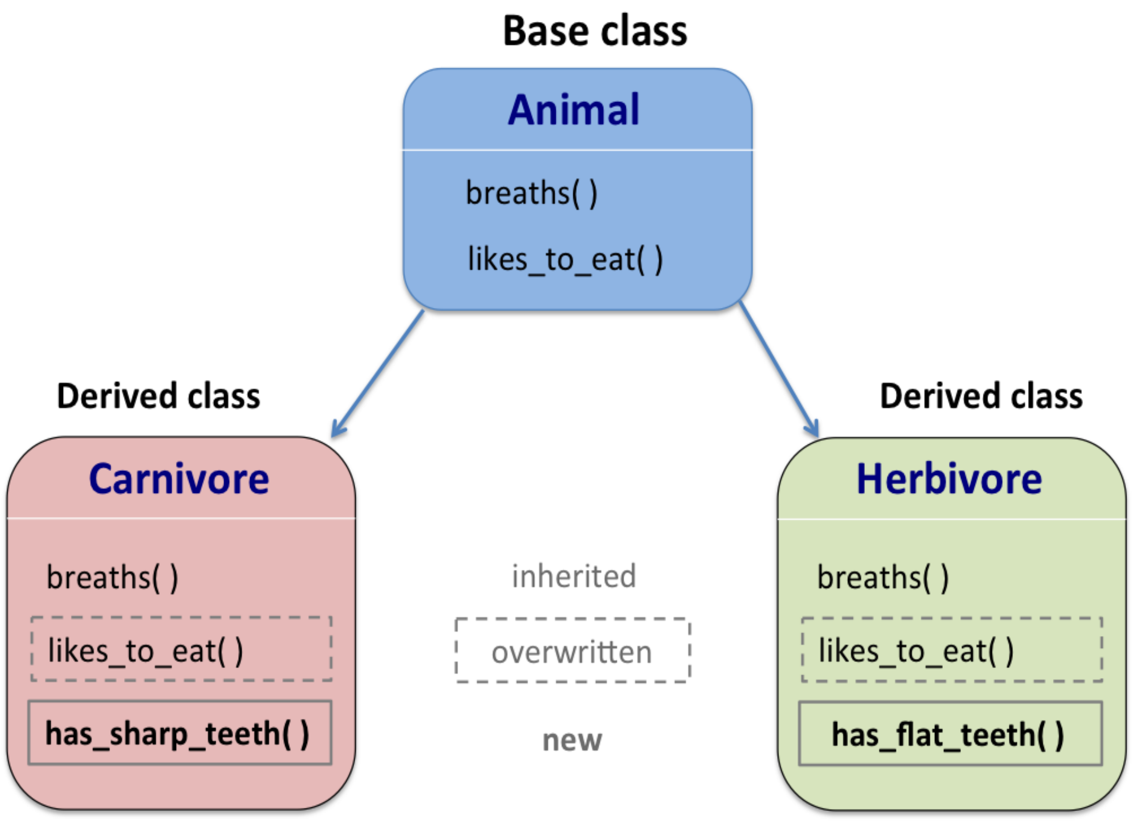


Fig 2. Schematic of inheritance. The animal, carnivore, herbivore example in the script. (These classes only have method attributes.)

2

If we then try to create some object and interrogate we find:

```
In [68]: ###first create an Animal and test it
In [69]: stuart = Animal()

In [70]: stuart.likes_to_eat()
Out[70]: 'food'

In [72]: stuart.breaths()
Out[72]: True


In [73]: ###now create a Carnivore and test it

In [74]: lion = Carnivore()

In [75]: lion.likes_to_eat()
Out[75]: 'Meat!'

In [76]: lion.has_sharp_teeth()
Out[76]: True



In [99]: ### Now create a Herbivore and test

In [100]: llama = Herbivore()

In [101]: llama.breaths()
Out[101]: True
```

```
In [102]: llama.has_flat_teeth()
Out[102]: True

In [103]: llama.likes_to_eat()
Out[103]: 'plants'
```

Important - Notice above that an object of the derived class (e.g. `Carnivore`) can use a method of its base class (e.g. `Animal`) in exactly the same way that an object of that base class does. This is illustrated in example above with:

```
stuart.breaths()     # This works as expected ...
lion.breaths()       # ... but so does this, thanks to 'inheritance'.
```

This interface to the objects is uniform, intuitive and natural, and makes sense because a `lion` *is an* animal! Having a uniform interface, as above, is known as *polymorphism* in computer science terminology.

Another important feature of OOP to note is that a derived class can itself act as a (immediate) base class for deriving a new class, and so on, thus creating an **inheritance hierarchy**. (See the presentation from Lecture 2, for diagrams illustrating repeated class derivation.)

Composition

Understanding the difference between *inheritance* and *composition* is important in OOP. You already used composition in worksheet 1, although it was not flagged up to you at that point. Composition is when an object contains other objects (i.e. the data attributes). Your **`FourVector`** class uses composition. Each `FourVector` object is composed of a float object storing *ct* and a numpy array that stores *r*. What is the difference between composition and inheritance?

- With composition, `object.method_of_a_data_attribute()` **does NOT work**!
- With inheritance, `object.method_of_base_class()` works, as seen already.

Let’s illustrate this using the **`FourVector`** class. Imagine you want to sum the space-like components. They are stored as a NumPy array, which conveniently has the method **`sum()`**:

```
>>> f = FourVector(ct=99, r=[1, 2, 3])
>>> f.sum()
Traceback (most recent call last):
  File "<ipython-input-93-6b42463c02e2>", line 1, in <module>
    f.sum()
AttributeError: 'FourVector' object has no attribute 'sum'

>>> f._FourVector__r.sum()    # Naughty!  Shouldn't be accessing hidden attributes!
6.0
```

`f.sum()` fails because a `FourVector` do not inherit any methods from its attributes `ct` and `r`, since `FourVector` is designed using composition. The second way achieves the desired goal by drilling down into the data attributes themselves. (Note how this is done by *chaining* attributes.) But accessing hidden attributes like this is considered bad practice and leads to fragile code!

Initialization functions under inheritance

If you want to add an `__init__()` method to your derived class you have to *explicitly* call the `__init__()` method of the base class. You can do this either by using the built-in function `super()` (not covered here) or by calling the constructor in a slightly unusual way. This is illustrated in the following example:

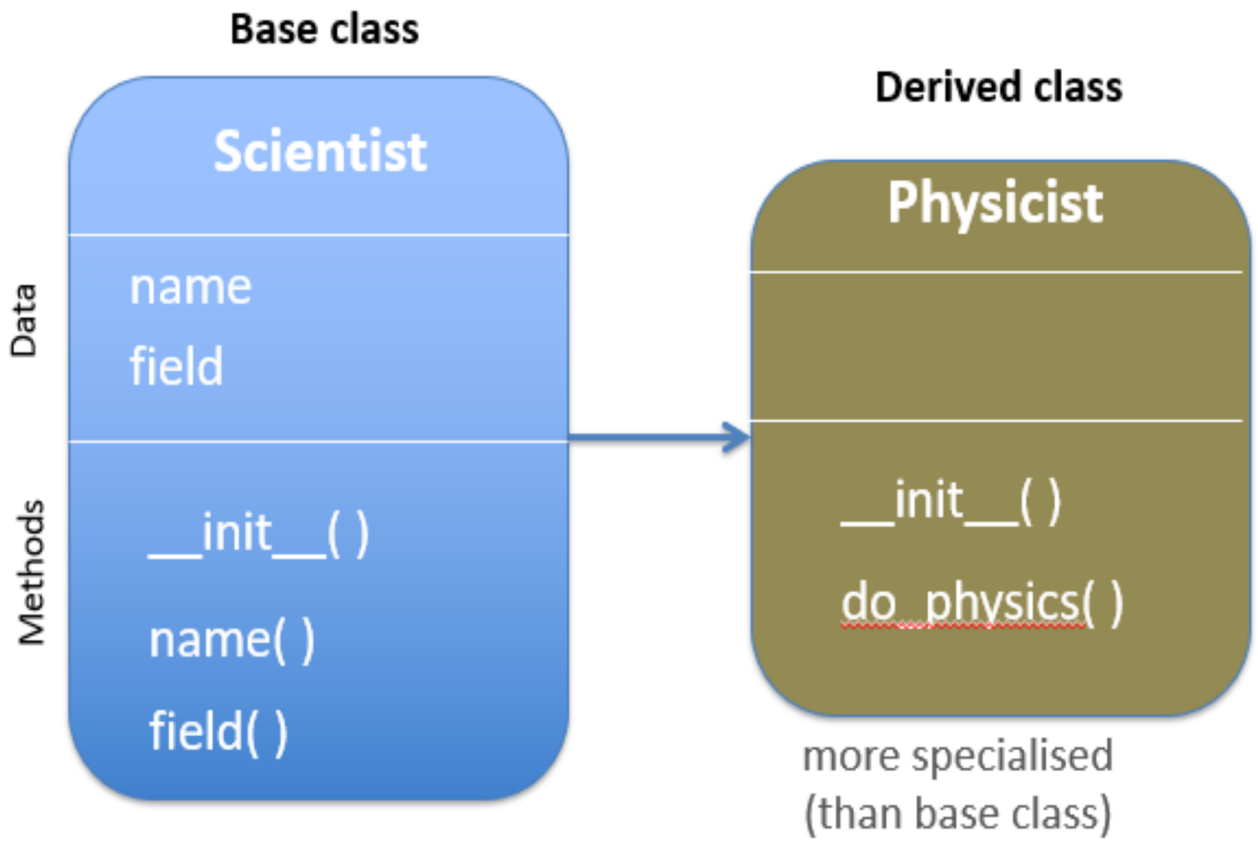
```
class Scientist:
    def __init__(self,name,field):
        self.__name = name
        self.__field = field

    def name(self):
        return self.__name

    def field(self):
        return self.__field


class Physicist(Scientist):
    def __init__(self,name):
        #Now the field is always "Physics"
        Scientist.__init__(self,name, "Physics")  #explicitly calling the Scientist
constructor
```

Diagrams for this example is shown in Figure 3.



This allows the following:

```
In [268]: John = Physicist("John")

In [269]: John.field()
Out[269]: 'Physics'

In [270]: John.name()
Out[270]: 'John'
```

which would not work if we had not explicitly invoked the base class' initialization function. (Note: This approach is more obvious than using `super()` when dealing with multiple inheritance. If you want to try the approach using `super()`, Google will provide lots of examples.)

(Be aware that methods in a class that create and initialize new objects are known as *constructors* in general OOP terminology. Thus the `__init__()` method in Python is a constructor. Similarly, methods that deal with deleting objects (not covered in this course) are categorised as *destructors* in OOP.)

Misuse of inheritance in OOP

Inheritance is often misused in OOP. The important rule to remember is that it is an **"is a"** relationship and not a **"has a"** relationship. A Carnivore **is an** Animal. It would be quite wrong to have a class Face that inherited from class Nose because a Face **has a** Nose rather than **is a** Nose. This may sound obvious but you would be amazed how often people make this mistake.

Composition is the appropriate OOP mechanism when a **has a** relationship exists between the new and existing objects. Thinking back to the `FourVector` class, a fourvector has a threevector (**r**) and a scalar (*t* or *ct*). But a fourvector is not a threevector. For example, the length is obtained in a different way and vector operations like the cross-product do not make sense for a fourvector. Similarly for our hypothetical Face class, composition from a nose, eyes, etc., would be appropriate.

Note: Rules in OOP:

- **Inheritance** is used when an **is a** relationship exists between the original type(s) and new type.
- **Composition** is used when a **has a** relationship exists between the original type(s) and new type.

If you think that the example above (and for that matter the task below) is rather contrived then you would be right. Inheritance is a very useful and powerful tool, however it is not always the right approach for directly simulating the physics of a situation. It is generally very useful in the construction of programs around the part where you simulate the physics.

Tasks on inheritance

Task 21: Consider that you are interested in coloured shapes on paper. Write a class `Shape` which has a hidden attribute of colour, an access method to get the colour (you may wish to have a default colour) and an modifier method to set the colour.

Try and make it so that you can **chain** your methods, e.g.

```
>>> s = Shape()
>>> s.set('Red').colour()
Red
>>> s.set('Yellow').set('Blue').colour()
Blue
```

Task 22: Now create three classes `Square`, `Triangle` and `Circle` which inherit from Shape. Each should have an `__init__()` method that specifies parameters useful for calculating the area of that particular shape.

Develop methods that return the area for each particular shape.

Then write specific examples where you set (and get) the colour of a triangle, square and circle as well return their areas.

This task demonstrates the important interface-like nature of the base class. Although your derived classes `Square`, `Triangle` and `Circle` did not themselves define :a colour access method they can happily utilise the one from their common base class `Shape`. In short, any class deriving from Shape will by virtue of this derivation be able to set and get it's own colour with no additional code necessary.

Stop 4: Check your work with a demonstrator for feedback.

isinstance()

Sometimes you need to check what sort of an object you have. The way to do this is to use:

```
isinstance(variable,Class)
```

As a derived class has an *is a* relationship with the base class isinstance will return true for a base class of a derived class.

Task 23: Use Shape and its derived classes to investigate the use of isinstance. What happens for variables that are type int or float?

One common use is to check that a function is being called with the correct objects. For example:

```
def func(a):
    if not isinstance(a, MyClass):
        raise TypeError("a needs to be an instance of MyClass")
```

Task 24: Write a function that will turn any Shape or it's derived classes "Red", but will not try to turn an integer (or indeed any other type of variable) red.

Class Variables

A *class variable* is a variable of the class (sounds circular but do read on) rather than a variable of the instance. The difference is that all instances have access to the class variables (if you use them). Another term for a class variable is *class attribute*. Consider the following:

```
class MyClass:
    # no need for dot notation
    class_var = 0
    def __init__(self, vv):
        self.inst_var = vv         # now need the dot notation
        MyClass.class_var += 1   # note that it is of MyClass not self
```

You should be able to see that class_var is acting as a counter and will count how many instances of MyClass actually exist. So for example:

```
In [1]: import class_var as cv

In [2]: a=cv.MyClass(4)

In [3]: # The instance variable should be 4
In [4]: a.inst_var
Out[4]: 4

In [5]: # When the instance a was initialised the class variable was incremented.
In [6]: cv.MyClass.class_var
Out[6]: 1

In [7]: # Note we can access the class_var through instance a.
In [8]: # This is discouraged however as it could be overridden by an instance variable with
the same name
In [9]: a.class_var
Out[9]: 1

In [10]: # Let's create more
In [11]: b=cv.MyClass(5)

In [12]: c=cv.MyClass(4.3)

In [13]: # The instance variables should be 5 & 4.3
In [14]: b.inst_var
Out[14]: 5
In [15]: c.inst_var
Out[15]: 4.3

In [16]: # The class variable will be incremented for each instance initialisation.
In [17]: cv.MyClass.class_var
Out[17]: 3

In [18]: # Likewise we can (but probably should not) access them through the instances a, b &
c
In [19]: a.class_var
Out[19]: 3

In [20]: b.class_var
Out[20]: 3

In [21]: c.class_var
Out[21]: 3
```

In this example the class variable is used as a simple counter, however in general it can be any type of variable and be used anywhere within the class. It should be fairly obvious therefore that this also provides a mechanism to change the behaviour of *all* instances of a particular class.

Another use for class variables is for properties that will be shared between different objects. For example in your shapes class you may have wanted to have pi as class variable.

Task 25: Modify your Shape class to count how many “Red” shapes that you have. Remember that if you have a “Red” shape that is set to be a different colour you have take this into account. Verify that this works and investigate what happens when you set the colours of the derived class. Is it what you would expect?

Stop 5: Check your work with a demonstrator for feedback.

Document history

Created: Oct 2013 - C Paterson

Revised: Oct 2017 - R Kingham

Revised: Oct 2020 - A Richards