Please refer to the online lesson plan on the DT Hub to access all website links and additional resources.

## Learning hook

Illustrate the basic concepts of OO programming by having students play Tetris – a game with which they are likely to be familiar.

1. Have the students play classic Tetris online.
2. After playing the game Tetris, have students participate in class discussion to determine:
* which different categories of objects might be created in designing this game?
(playing board, playing pieces, high score gallery)
Explain that we call these *classes*in OOP.
* what would be the *attributes*or *properties*of each of these these classes?
(playing board: different colours, dimensions, music choice etc)
(playing pieces: 7 different shapes, colour, rotation)
* what *methods*or *behaviours*might be invented for these classes?
(playing board: gravity)
(playing pieces: falling, moving sideways and rotating)
* one of our classes is the family of playing pieces. What different ones might be created?
(One playing board as an instance of the Playing board class; 7 differently shaped blocks as instances of the Playing pieces class, called Tetrominos, which are the required playing objects)
In OOP these are called different *instances*of the ‘playing piece’ class.

**Car company analogy**

The analogy of a car company producing different models in their factory may assist in explaining OOP principles and advantages.

Each model of car designed by a car company can be considered a separate class described by the factory’s blueprints for each class, designed before any of the cars exist.

One of these classes might be a hatchback design. This class will have certain properties, such as colour, hatchback door, airconditioning (yes or no), luxury additions (yes or no) and so on. It will also have certain behaviours that it owns such as a hatchback door that can be lifted up.

When the first hatchback is made, it is said to be an instance of the hatchback class. All new hatchbacks will share the same properties and behaviours designed for the hatchback class.

There will be other classes too, such as a sports model, with its own properties and behaviours.

In fact, it might be a good idea to create a kind of ‘super class’ called ‘car’ to which all these other classes belong and assign to it all the properties and behaviours that all models possess, to save us repeating them every time a new one comes out. After all, all our models will have headlights, steering wheels, seats and engines!

When we design a complex program, it is much easier if similar objects are treated as classes, with properties and behaviours. Once we have done this we don’t need to keep thinking about the details, as we have made a component we can use when we need to, knowing it will always behave the same way.

Functions perform a similar role in programming as we write them once and just call them when we need that behaviour, but OOP takes that a step further.  Instead of having lots of different functions scattered through our programs, making it easy for a programmer to get lost, OOP collects all functions belonging to the same object and groups them as a class.

## Learning map and outcomes

Students learn about Object Oriented Programming (OOP) methods using classes as a useful approach in the development of more complex programs, where separate objects are defined as having attributes and behaviours.

## Learning input

**Text based programming approach: teaching Python using OOP principles**

This approach is suitable for more able students. In this lesson you introduce OOP principles using a text-based OOP programming language with which students are familiar.

Without using actual programming code, text-based programming using OOP can only be explained in the most cursory manner. We thus have the difficulty of choosing a computer language for the purposes of illustrating classes, attributes and behaviours.

Explain to students that they will be using Python in this example: it is an OOP language, provides a good springboard for learning other languages, is widely used in schools, is well supported and is freely available.

**Building a Role playing (RPG) game following OOP principles using classes**

Tell students: If you were asked to create a RPG game at this stage of your programming experience you might design it in this way:

1. By simply using many variables:

heroName = 'Elsa'
heroHealth = 50
heroMagicPoints = 80
heroInventory = {'gold': 40, 'healing potion': 2, 'key': 1}
monsterName = 'Goblin'
monsterHealth = 20
monsterMagicPoints = 0
monsterInventory = {'gold': 12, 'dagger': 1}

1. And if you wanted to add more monsters, say a Dragon and another Goblin, you could use a list structure such as that found in the Python language [….]:

monsterName = ['Goblin', 'Dragon', 'Goblin']
monsterHealth = [20, 300, 18]
monsterMagicPoints = [0, 200, 0]
monsterInventory = [{'gold': 12, 'dagger': 1}, {'gold': 890, 'magic amulet': 1}, {'gold': 15, 'dagger': 1}]

All you would need to use to refer to a particular monster would be its index: from 0 to 2.

However, there would be a problem if just one of these values entries from your lists was deleted. All the values in that list would then be reassigned to the wrong monster.

1. A better way is to create a single blueprint for all monsters, and since they share features with other living things, such as your hero, you could bundle them all into one class and call it LivingThing(). Every time you wanted to create a creature, you could just call this as your constructor, giving it its initial values.

In Python it might look something like this (we will explain all separate the parts of this next!):

class LivingThing(object):
    def \_\_init\_\_(self, name, health, magicPoints, inventory):
        self.name = name
        self.health = health
        self.magicPoints = magicPoints
        self.inventory = inventory

* 1. Create the LivingThing objects for the hero and the monsters. First the hero:

hero = LivingThing('Elsa', 50, 80, {})

* 1. Now the monsters. Create a list to store the monsters, each one created from your class definition:

monsters = []
monsters.append(LivingThing('Goblin', 20, 0, {'gold': 12, 'dagger': 1}))
monsters.append(LivingThing('Dragon', 300, 200, {'gold': 890, 'magic amulet': 1}))

We have halved the amount of code we needed to write as we are using the same code for both hero and monsters.

**Dissecting our class structure**

Let us examine each part of our **class**code.

1. We first defined a new class and called it LivingThing:

class LivingThing(object):

* + We started the class definition with the class keyword, followed by the class name and a colon. We would list any parent classes in between round parentheses before the colon, but this class doesn’t have any, so we can leave them out.
	+ In Python everything is an object. Note the word object in parentheses? We could omit it and just write class LivingThing(): and it would have no effect here.
1. After this we defined a function, also called a **method**, which is just a name used for functions that belong to a class:

 def \_\_init\_\_(self, name, health, magicPoints, inventory):

* + def here is short for ‘define’. It is saying that what follows is a definition of something. In this case it is a class object. It is a blueprint for the objects it will create and not the objects themselves.
	+ \_\_init\_\_( ) is a special method. When we call the class object, a new instance of the class is created, and the \_\_init\_\_ method for this new object is immediately executed with four attributes belonging to our class object: name, health, magic points and inventory. The purpose of this \_\_init\_\_() method is to create up a new object using the data that we provide. Our LivingThings receive four attributes when it is first created.
1. We now define the variables needed for each instance of LivingThings which we create:

 self.name = name self.health = health self.magicPoints = magicPoints self.inventory = inventory

This creates these four class member variables for the object being created. They all begin with self to show they are member variables belonging to our instances and not just local variables which would disappear after the class is used.

1. The last part shows how these four variables are assigned their initial values:

hero = LivingThing('Elsa', 50, 80, {})
monsters = []
monsters.append(LivingThing('Goblin', 20, 0, {'gold': 12, 'dagger': 1}))
monsters.append(LivingThing('Dragon', 300, 200, {'gold': 890, 'magic amulet': 1}))

* + This is where we finally get to utilise the constructor we have defined for our class by ‘calling it’. So when we write:
	LivingThing()
	we are calling the LivingThing class's \_\_init\_\_() method, its constructor.

LivingThing('Elsa', 50, 80, {})

thus creates a new LivingThing object (our hero!) which it stores in a variable called hero. (Note that at this stage Elsa has been given no weapons!)

* + Next we define a new empty list and call it monsters.
	Now we create two more LivingThing objects, in this case two monsters, and store them in our new list called monsters. Notice that, as well as being given a name, age and magic points, our two monsters each carry two items.
1. If we need to print these details we would use something like the following – notice our hero is just one item and so we can access each variable by using dot notation:
print('Our hero is',hero.name, 'and she faces a:')
Our monsters are stored a list structure, so we need to identify which monster in the list we are referring to by using its index n (which always starts at 0).
Len(monsters) finds out how many items are in the monsters list and then we move through the list from the first to the last one:
for n in range(len(monsters)):

print(monsters[n].name)
print('with health of',monsters[n].health)
print('and magic points',monsters[n].magicPoints)
print('and these objects',monsters[n].inventory)
print()

**Benefits of using classes**

Here's where we begin to see the benefits of object-oriented programming. If another programmer is reading your code, when they see LivingThing() they know they can just look for the "class LivingThing" and find out the details of everything they need to know about what a LivingThing is.
Furthermore, they know they can update the LivingThing object and all objects created from it will now gain that new feature. This is much easier than trawling through a long program trying to find what things need to be changed.

## Learning construction

Having explained and illustrated the concept of a class and the rules for writing its definition, introduce the Learning construction activity: creating a Birthday Book program using OOP principles.
It is possible to have students develop this entire code themselves, in concert with a teacher in the role of ‘Socratic guide’.
However, the approach taken here is that of a ‘guided construction’. Your role is to seek student responses to sections printed in green and assist where necessary and to provide students with those parts of the code printed in black, accompanied by an explanation.
Note the following:

* A text file must exist and be place in the same directory as the working Python program.
* Students should never copy and paste code but should be required to type it out. A critical part of learning to code involves learning the importance of precision. In addition, students remember and learn better when are required to complete steps themselves.

There are three stages, with the third being optional.

* In Stage 1, students create their own Birthday Book program by completing the annotated missing parts of a Python program first introduced by you.
* In Stage 2, students make improvements to their working code from Stage 1.

**Stage 1: Building the basic program**

At the end of this stage students make important suggested improvements to the program. Explain the following steps to the students.

1. Create a class for each person’s entry:

class BirthdayEntry():
def \_\_init\_\_(self, given\_name = None, family\_name = None, email\_address = None, DOB = None):

1. Add variable names used by instances of this class:

self.given\_name = given\_name
self.family\_name = family\_name
self.email\_address = email\_address
self.DOB = DOB

1. Also add a method for formatting display of items. This can be explained and provided for students:

def display(self):
print('{} {}: {} {}'.format(self.given\_name, self.family\_name, self.email\_address, self.DOB))

1. Create a class to hold all the entries using a list data structure.
Since we made it a class, we could create other address books for use in future without needing to rewrite the code. Also we add a method of displaying.

class BirthdayBook():
def \_\_init\_\_(self):
self.friends=[]
def add\_newentry(self, new\_entry):
self.friends.append(new\_entry)
def display(self):
for friend in self.friends:
friend.display()

1. Create a class called UserInterface to run everything.

	1. We write a class that creates an instance of our BirthdayBook class and calls a method which delivers choices to the user such as adding an entry or displaying the complete book (the choices method will be written later).

class UserInterface():
def \_\_init\_\_(self,book):
self.birthday\_book=book
self.choices()

* 1. Add to this class a method which allows new entries to be added:

def add\_entry(self):
print ("Adding a new person")
given\_name =input("Given Name? ")
family\_name =input("Family Name? ")
email\_address =input("Email address? ")
DOB\_PROMPT="Date of Birth in form DD MM YY? "
DOB=input(DOB\_PROMPT)

* 1. We now call this entry birthday\_entry and add it to our birthday book:

birthday\_entry=BirthdayEntry(given\_name, family\_name, email\_address, DOB)
self.birthday\_book.add\_newentry(birthday\_entry)

* 1. We next create our Choices method which is added to the Engine class:

def choices(self):
choice=input("Make a selection from the above")
if choice=="a":
self.add\_entry()
elif choice=="i":
print("Students devise Instruction menu")
elif choice =="d":
self.display\_summaries()

* 1. We write a method to display our birthday book:

def display\_summaries(self):
self.birthday\_book.display()

* 1. Finally, we make the whole program run simply by firing up our UserInterface class after we create an instance of our BirthdayBook:

book = BirthdayBook()
UserInterface(book)

For testing purposes it might be helpful to ‘hard-wire’ two entries and place them between these two lines to provide data to be able to test our display option:

book = BirthdayBook()
book.add\_newentry(BirthdayEntry('Bob', 'Smith', 'bsmith@python.com', '11111'))
book.add\_newentry(BirthdayEntry('Mary', 'Smith', 'msmith@python.com', '22222'))
UserInterface(book)

Also for testing purposes it will be helpful to place the program inside a loop to allow repeated entries to be added.

* 1. A working Python file is provided showing the final outcome of this stage ( birthdaybook\_barebones (PY)).
	A more refined version is also provided (birthday\_book\_1 (PY)).

**Stage 2: Making improvements to the program**

A working Python file provided shows the final outcome of a few of the suggested improvements for Stage 2: birthdaybook\_2.py
Encourage students to work alone to achieve the following refinements once their bare-bones code from Stage 1 is working:

* Add internal comments throughout the program
* Ask user if they wish to continue or quit
* Require email to be valid (contain a ‘@’ symbol in it)
* For each entry variable offer a way of escape while entering data
* Improve the Instruction menu appearance
* Add response if invalid entries are made
* Improve line spacing throughout output

More advanced improvements:

* Require a proper date format using the language’s date syntax
* Add ability to edit or delete stored entries
* Students suggest other improvements

**Stage 3: Writing to and reading from external files (optional)**

You may wish to have students learn to write to and read from an external file.
An example program is provided with external text file (ensure these two files are in the same directory):

* Birthday\_book\_2 (PY)
* workfile (TXT)

## Learning demo

Students demonstrate their working Birthday Book code and show a feature they have developed themselves.

## Curriculum links

| Links with Digital Technologies Curriculum Area |
| --- |
| **Strand** | **Content Description** |
| **Processes and Production Skills** | Implement, modify and debug modular programs, applying selected algorithms and data structures, including in an object-oriented programming language (AC9TDI10P09). |

## Assessment

|  | **Quantity of Knowledge** | **Quality of Understanding** |
| --- | --- | --- |
| **Game creation** | No evidence of understanding | Student is able to build classes for the design | Student is able to build correct classes for the design | Student is able to integrate the classes into a single correct working program | Student is able to add at least one correctly working feature to their program |
| **Optional Score** | 0 | 1 | 2 | 3 | 4 |