Classes

One of the most powerful ways of organising code is through the use of classes. These are a key concept across programming languages, and, once you make sense of exactly what they are, can revolutionize how easily you can make sense of, and work with, increasingly complex pieces of code.

A class is:

- ***The blueprint for a new custom object.***
- ***A convenient way to hold variables and functions that relate to a specific object, and are common to all such objects.***

So what *is* an object in Python?

Wrong question. To see why, try completing the following sentences:

A smartphone is ____
____

Multiplication is ____
____

My guess is, your descriptions hinged pretty heavily on what a smartphone *does* rather than some abstract way of describing what it *is*.
With multiplication, it's even more fundamental: it is defined by what it *does*.

Our brains are hard-wired to filter out almost all sensory input we receive (you could consciously focus your attention on the position of your left foot, or the feel of the chair back, or the hum of the computer monitor, or the texture of the desk surface, but you - generally - don't). We see threats and opportunities in our environment, and in particular those things which we have a name for. Children can't even distinguish between blue and green, or between red and orange, until we give them names and repeatedly identify them as distinct 'objects' or ideas. Rather than ask "what is it?", ask "what does it do?"

OK, so what does an object in Python *do*?

You decide. That's the beauty of custom objects. Whoever invents the concept of a particular object decides exactly how it behaves. Someone invented the **Turtle** object, which has attributes (properties) such as size, shape and position, and methods (functions specific to the object) such as turn, forward, penup, etc. You have used the built-in Python object **int**, which is designed to behave as we would expect integers to behave. They have certain properties, such as size, and we can do certain things with them, such as increase them by 1, or add one to another to create a third.

A real-world example

You visit the states, and run into someone who doesn't know what a kettle is (most houses have a coffee machine, but tea isn't quite so popular across the pond). So you want to explain what one is. Kettles are defined primarily by what they do, so there are things that resemble kettles which aren't, and things which don't resemble kettles which are:

Just as you would likely describe a kettle by describing **what it has** and **what it does**, all objects in Python are defined by their **attributes** and their **methods**. See if you can add to the list below.

Type of object: Kettle

Attribute
Capacity: The maximum amount of water it can hold, in litres.
Power: The power output in Watts.
Water: The amount of water it currently contains.
Running: Indication of whether it is currently heating the water (True or False).
Temperature: The current temperature, in $°C$, of the water.

Method
Boil: If the kettle is not currently running, turn it on.
Shut-off: Check to see if the temperature is above a certain threshold (eg $90°C$), and if so, shut off the power.
Refill (amount): Top up the water by the amount required, or until it reaches maximum capacity, whichever comes first.
Pour (amount): Remove the amount required, or all of the water, whichever comes first.

Tip: Working out the attributes and methods of a Python object before coding it up is a very good idea.

Notation: creating a class

Python has a specific way of creating objects. We use the key word **class** to indicate that we are creating the blueprint (design specs) for a new type of object, then we set up its initial properties in a specially named **__init__** function (two underscores either side):

```
class Kettle:
    """Kettle object takes arguments capacity (l) and power (W)"""
    def __init__(self, capacity, power):
        self.capacity = capacity
        self.power = power
        self.water = 0
        self.minimum = 0.2
        self.running = False
        self.temperature = 10

    def boil(self):
        """Run kettle while water level high enough and not boiling"""
        if self.water >= self.minimum:
            self.running = True
        else:
            self.running = False
        self.temp_shut_off()

    def temp_shut_off(self):
        """Check temperature, and shut off if boiling"""
        if self.temperature >= 90:
            self.running = False

    def refill(self, amount):
        """Top up by given amount, or up to capacity"""
        if self.water + amount > self.capacity:
            self.water = self.capacity
        else:
            self.water += amount

    def pour(self, amount):
        """Remove given amount of water, or all of it"""
        if self.water > amount:
            self.water -= amount
        else:
            self.water = 0
```

The first line uses **class** and then the name of your object (capitalized by convention).

The first function you define, **__init__**, initializes (sets up) any kettles you create.

The arguments for the **__init__** function start with the key word **self**. This is like a hidden argument, since when this blueprint is used to generate a kettle, we would only need to provide it with **capacity** and **power**.

The **self** keyword appears at the start of every method within the function, indicating that the function is specific to this particular object. Also, variables that are specific to this object are preceded by **self.** so that Python knows which particular kettle they relate to.

Notation: creating specific instances

```
my_kettle = Kettle(1.5, 1200)
print(f"My new kettle has capacity {my_kettle.capacity} litres.")
print(f"It's got {my_kettle.water} litres in it.  Refilling...")
my_kettle.refill(2)
print(f"My kettle has {my_kettle.water} litres of water.")
print(f"Current temperature: {my_kettle.temperature}.")
my_kettle.boil()
print("Boiling...")
while my_kettle.running:
    my_kettle.temperature += 1
    my_kettle.temp_shut_off()
print(f"Temperature: {my_kettle.temperature}.  Pouring...")
my_kettle.pour(0.4)
print(f"Done.  Currently holds {my_kettle.water} litres of water.")
```

Writing the code for the class doesn't actually create any kettles (not even any virtual kettles!) – to do that, we have to make a new variable, and tell Python to make a kettle using our **Kettle** object. We can get its attributes using the dot notation (eg **my_kettle.water**), and run its methods similarly (eg **my_kettle.refill(2)** or **my_kettle.boil()** etc)

Class, or instance of a class?

It is important to distinguish between a **class** and a specific **instance** of that class.
Using our **Kettle** class as an example, the **class** is the blueprint which sets out for Python what it means to be a kettle – it's a collection of the various attributes and methods that *any* specific kettle we choose to make should have.
If we actually make a particular kettle now, using the blueprint provided by the class code, that kettle would be an **instance of the class**. The process of making a particular kettle is described as "instantiating an instance of the Kettle class". We would then have a particular kettle, named whatever we like, which automatically has all the attributes and methods that we have previously decided any kettle should have.

*A nice analogy is **elements** and **atoms**:* a carbon atom is a specific object which shares certain properties in common with all such atoms. We classify these atoms based on their common properties and behaviours, and the name given to this type of atom is **Carbon** (an *element*). When we say things like "Carbon can bond with oxygen", we aren't referring to any one specific carbon atom, but describing a common property of all carbon atoms. If we say "To make a molecule of carbon dioxide we're going to need one carbon and two oxygens", we are referring to *instances of the carbon class*: actual atoms.

```
class Carbon:
  def __init__(self):
    self.atomic_number = 6
    self.protons = 6
    self.neutrons = 6
    self.electrons = 6
    self.symbol = "C"
    self.atomic_weight = 12.0107
    self.temperature = 10
    self.state = "solid"
    self.melting_point = 3550
    self.boiling_point = 3800
```

This first block of code tells Python what we mean by a **Carbon** object. It is initialised without any arguments (apart from the default hidden **self** key word), and attributes are automatically assigned.

```
  def decay(self):
    if self.neutrons == 7:
      if random.random() < 0.0001:
        self.neutrons = 6

  def heat(self, increase):
    self.temperature += increase
    if self.temperature > self.boiling_point:
      self.state = "gas"
    elif self.temperature > self.melting_point:
      self.state = "liquid"
    else:
      self.state = "solid"
```

The class currently supports two methods: you can make your atom **decay** (with probability 0.01%) if it currently has more neutrons than normal, and you can change the temperature of your atom using **heat**. This method also updates the **state** of the atom (solid, liquid or gas).

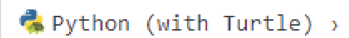
```
glucose = []
for i in range(6):
  new_atom = Carbon()
  glucose.append(new_atom)
for i in range(12):
  new_atom = Hydrogen()
  glucose.append(new_atom)
for i in range(6):
  new_atom = Oxygen()
  glucose.append(new_atom)

for atom in glucose:
  print(atom.symbol)
```

Since classes are such a powerful way of making new objects of the same type very easily, we often don't even bother naming them. In the snippet on the left, we create 6 new atoms using the **Carbon** blueprint, and put these six objects in a list (**glucose**). A Python list can contain any Python object, including those we design ourselves, and, as the last lines show, we can loop through the list in order to do things with each element.

Next, complete the *Rectangle Class Challenge*, making a Rectangle object in Python…

Making an object *with Python*

For this project, if you're using repl.it, you'll need to choose:



We're going to start by creating an object which should share the same properties and functionality as the abstract mathematical concept: the *rectangle*. Individual rectangles may differ in various ways, but all rectangles share certain common properties, and we're going to create a class in Python that dictates what it means to be a rectangle.
The best way to do this is to think about what you might want to do with the objects once you have a way of creating them. Let's say I'm interested in drawing lots of rectangles on the screen, with various dimensions and in various positions. Maybe I want to draw a maze, or build some interesting patterns, or display all rectangles with a certain perimeter.

Step 0: Figure out what you want your object to do. What is it for, and what would you like to be able to get from it? For this example, I'll define the specification for you:

Attribute
w: The width of the rectangle.
h: The height of the rectangle.

Method
area: Returns the area of the rectangle.
perimeter: Returns the perimeter of the rectangle.
diagonal: Returns the length of a diagonal across the rectangle.

Step 1: Give your class a name, and make the __init__ method:

```
class Rectangle:
    """A rectangle with width w and height h"""
    def __init__(self, w, h):
        self.w = w
        self.h = h
```

Start with the key word **class**, then the name of the object you are creating, capitalized (**Rectangle**), then a colon (**:**)

The next line, enclosed in triple speech marks (**"""**), is the **metatext** – it's a bit like a comment, but one which defines (for both the author and the user) what the function does.

The **__init__** function initializes any new object made using these specifications, so use it to create the object-specific attributes: make the object's width equal to the value of **w** that will be given when a new rectangle is created.

Step 2: Write the class methods:

```
    def area(self):
        """Returns the area of the rectangle"""
        return self.w * self.h
```

The **area** function opposite, provided it is defined within the **Rectangle** class (indented in line with the **__init__** function), is a class method.

Now write methods for the other methods described in the specification above.
You can test your code by making a bunch of rectangle objects to investigate.

Turn over to see how to develop this object so we can display rectangles on screen…

Step 3: Adapt and extend:

```
1   import turtle
2
3   class Rectangle:
4     def __init__(self, x, y, w, h):
5       self.x = x
6       self.y = y
7       self.w = w
8       self.h = h
9
10    def draw(self):
11      t = turtle.Turtle()
12      t.hideturtle()
13      t.penup()
14      t.speed(0)
15      t.goto(self.x, self.y)
16      t.pendown()
17      t.goto(self.x + self.w, self.y)
18      t.goto(self.x + self.w, self.y + self.h)
19      t.goto(self.x, self.y + self.h)
20      t.goto(self.x, self.y)
21
22  rectangles = []
23  for i in range(10):
24    new_rectangle = Rectangle(i * 20, -i * 10, 20, 30)
25    rectangles.append(new_rectangle)
26
27  for rect in rectangles:
28    rect.draw()
```

If I want to display rectangles on the screen, I need information about *where* the rectangle is as well as its size.

In this version, I have added **self.x** and **self.y** attributes, and also made use of the **turtle** module to draw my rectangle on the screen when requested.

The **draw** method I have written creates a turtle object, putting it in the bottom-left corner before drawing with it. Then it draws all four sides.

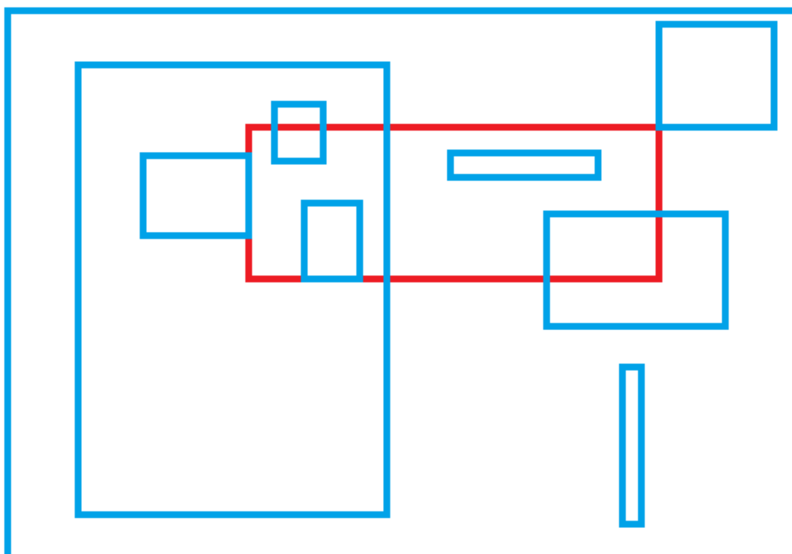
The next lines are where actual rectangle objects are created. Because I want to make lots, I have started with an empty list called **rectangles**. I then loop through, making new rectangles (I've set it up so each is in a different position – you can also import the **random** module and put them all over the place).

Finally, I loop through the list, drawing each rectangle using its **draw** method.

Step 4: Play

Try to come up with some ideas for Rectangle methods yourself.

- Can you use the **fill** functions of the **turtle** module to make them different colours?
- Can you make a program to draw a chessboard with different shadings of squares?
- Can you make a method for moving the rectangle?
- Can you build a method that tests to see if a rectangle overlaps with another?
-



Use this diagram to determine what conditions would need to be satisfied for a rectangle to overlap or not overlap with another rectangle. Which of these blue rectangles should count as 'overlapping' with the red rectangle?