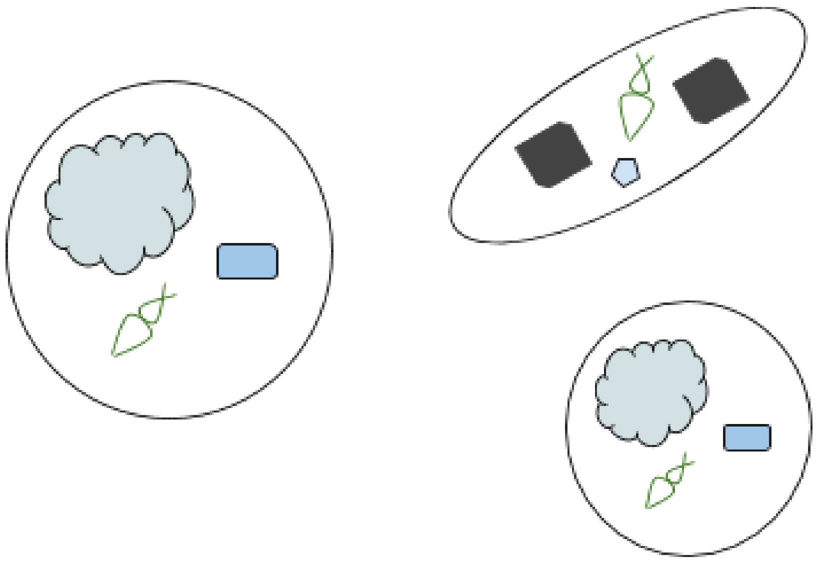
Week 5 - Object Orientated Programming

This lab sheet will introduce a very important concept of computer science: object orientated programming. After this session you will be familiar with the following concepts:

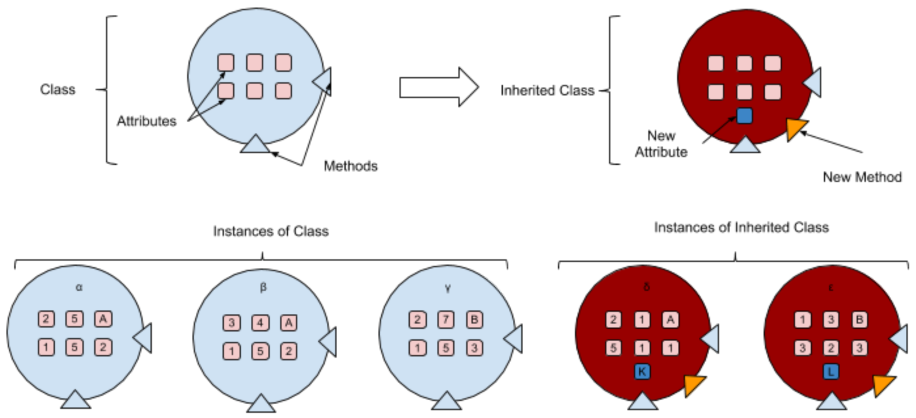
- Classes;
- Attributes;
- Methods;
- Inheritance.

Classes

1. **TICKABLE** The main idea behind object orientated programming (OOP) is to create abstract structures that allow us to not worry about data. Alan Kay come up with the concept and is quoted as saying: 'I wanted to get rid of data'. Instead of keeping track of variables using lists and arrays and writing specific functions for each operation we could be trying to do we use a system similar to the cellular structure in biology:



The following image shows the various ideas that we will work through in this lab sheet (keep it in mind as we go through things).



Here we will see how to create our own **class** (a type of object). The following code creates a class called `Student`:

```
class Student()
    pass
```

The `pass` command is just a line of code to say that this class does not actually do anything (we'll see much more exciting things soon). **Importantly**, this does not create an actual object. To create an **instance** of a class we assign it to a variable name:

```
vince = Student()
```

To view the object try the following:

```
print vince
```

Create a nother instance of the Student object and view it.

Note the convention used here: classes use a capitalized name.

2. Discussing other classes

   We have already used various types of python objects:

   - Lists;
   - Dictionaries;
   - Integers;
   - Strings…

   When we have converted a numeric variable to a string:

```
a = 25
a = str(a)
```

We have in fact created a new object of the string class and assigned that to the variable `a`.

Attributes

3. So far we have done nothing interesting. We now see how to make our objects ‘hold’ information. The following code re-creates our previous student class but gives the class some ‘attributes’:

```
class Student():
    courses = ["Biology", "Mathematics", "English"]
    age = 12
    sex = 'Male'
```

If we now assign the variable `vince` to an instance of Student we can access these attributes as follows:

```
vince = Student()
print vince.courses
print vince.age
print vince.sex
```

We can then modify these attributes like any other python object:

```
vince.courses.append("French")
print vince.courses

vince.age = 28
print vince.age

vince.sex = "M"
print vince.sex
```

4. **TICKABLE** A quadratic is a mathematical expression of the form: $ax^2 + bx + c$. The following code creates a Quadratic class:

```
class Quadratic()
    a = 0
    b = 0
    c = 0
```

The file W05_D01.csv contains triplets of coefficients. Create instances of the Quadratic class with the corresponding coefficients and put them all in to a list:

```
listofquadratics = []
```

Methods

5. **TICKABLE** We have seen methods before. For example for a given list `l = [1, 3, 2, 1.2]`, there exists a `sort` method that 'does some work to l' so that `l` becomes: `[1, 1.2, 2, 3]`. It is very easy to create custom methods for any class. We simply need to define functions *within* our class.

   Let us revisit the Student class but we now include a method `haveabirthday` which increments the age:

```
class Student()
    courses = ["Biology", "Mathematics", "English"]
    age = 12
    sex = 'Male'

    def haveabirthday(self):
        self.age += 1
```

   All method definitions require `self` as an argument which simply implies that that method work on the particular instance in question. We do not need to pass an argument to methods for the self argument (we can in essence ignore it).

   Let us see how this works:

```
vince = Student()
print vince.age
vince.haveabirthday()
print vince.age
```

   We can include other arguments in a method definition (just like for a normal python function):

```
class Student()
    courses = ["Biology", "Mathematics", "English"]
    age = 12
    sex = 'Male'

    def haveabirthday(self, numberofbirthdays=1):
        self.age += numberofbirthdays
```

In the above we have set a default variable:

```
vince = Student()
print vince.age
vince.haveabirthday()
print vince.age
vince.haveabirthday(28)
print vince.age
```

6. There are a variety of 'special' methods to be defined no classes. The one we will consider here is the `__init__` method. This method is fun when an instance is created. The following code allows us to pass arguments to an instance of a class when it is **initialised** to set attributes.

```
class Student():
    def __init__(self, courses, age, sex):
        self.courses = courses
        self.age = age
        self.sex = sex
    def haveabirthday(self, numberofbirthdays):
        self.age += numberofbirthdays

vince = Student(["Biology","Math"], 28, "Male")
print vince.courses
print vince.age
print vince.sex
```

7. **TICKABLE** Re visit the Quadratic class from earlier. Use the `__iter__` method and also create a method to return `True` if the quadratic has real roots.

   Recall that the roots of $ax^2 + bx + c$ are given by:

$$x = \frac{-b \pm \sqrt{b^2 - 4a}}{2a}$$

   Using this count how many of the instances in W05_D01.csv have real roots.

8. The following dictionary contains course code - course name pairs:

```
coursedictionary = {1: "Math",
                    2: "English",
                    3: "French",
                    4: "Physics",
                    5: "PE",
            6: "Biology",
            7: "History",
            8: "Geography"}
```

The following list contains lists containing a student name as well as a list of the course codes on which they are enrolled.

Using object orientated programming obtain the number of students enrolled on each course as well as a dictionary containing students as keys and lists of enrolled courses as values.

9. If rain drops were to fall *randomly* on a square of side length $2r$ the probability of the drops landing in an inscribed circle of radius r would be given by:

$$P = \frac{\text{Area of circle}}{\text{Area of square}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

Thus, if we can approximate P then we can approximate π as $4P$. In this question we will write code to approximate P using the random python library.

First of all, create a class for a rain drop (make sure you understand the code!):

```
class Drop():
    def __iter__(self, r=1):
        self.x = (.5 - random.random()) * 2 * r
        self.y = (.5 - random.random()) * 2 * r
        self.incircle = (self.y) ** 2 + (self.x) ** 2 <= (r) ** 2
```

Note that the above uses the following equation for a circle centred at $(0, 0)$ of radius r:

$$x^2 + y^2 \leq r^2$$

.

To approximate P simply create N=1000 instances of Drops and count the number of those that are in the circle.

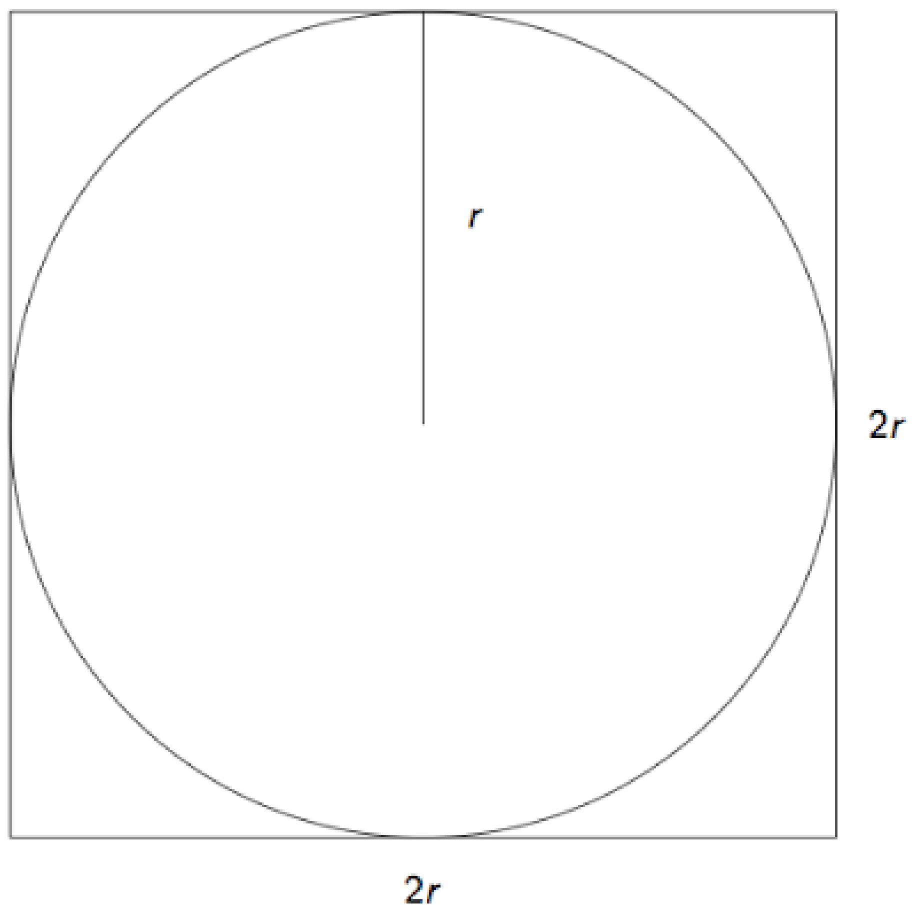
Use this to approximate π. What happens if you increase N?

(This is an example of a technique called Monte Carlo Simulation.)

Inheritance

10. **TICKABLE** One final important concept of object orientated programming that we will consider is ‘inheritabce’. This allows us create hierarchical structures of classes where some classes are derived from others.

Recalling the Student class:


r
$2r$
$2r$

```
class Student():
    def __init__(self, courses, age, sex):
        self.courses = courses
        self.age = age
        self.sex = sex
    def haveabirthday(self, numberofbirthdays):
        self.age += numberofbirthdays
```

We can create a new class from the Student class:

```
class MathStudent(Student):
    favouriteclass = "Mathematics"
```

Instances of this inherited class will inherit attributes and methods from the original class:

```
becky = MathStudent(['Mathematics', 'Biology'], 29, 'Female')
print becky.courses
print becky.age
print becky.favouriteclass
beck.haveabirthday()
print becky.age
```

Create another inherited class (from student) giving it a specific method.