# Unit 8. Lecture A

# Object-Oriented Programming (OOP II)

**Dr Sofiat Olaosebikan**

School of Computing Science

University of Glasgow

CS1P. Semester 2. Python 3.x

Object-Oriented Programming (OOP) is a programming paradigm that revolves around the concept of objects. Objects are instances of classes, and classes are user-defined data types made up of attributes and methods.

OOP is built upon four core principles:

1. **Encapsulation**: This principle bundles attributes and methods into classes. It also *promotes data integrity and security* by restricting direct access to internal data
2. **Inheritance**: It allows creating new classes (subclasses) that inherit properties and behavior from existing classes (superclasses). This *reduces code duplication* and allows us to create a hierarchy of classes.
3. **Polymorphism**: It allows us treat objects of different classes as objects of a common base class. It promotes *adaptability* and *maintainability*.
4. **Abstraction**: It simplifies complex systems by modeling classes based on essential properties and behaviors It hides the underlying implementation details and promotes code *reusability* and *maintainability* by separating the "what" from the "how."

So far, we have seen:

- how to define a class
- instance and class variables
- instance and class methods
- string representation of an object using `__str__` and `__repr__`

## Outline

In this lecture, we will cover:

- operator overloading
- inheritance

### Applying arithmetic operators on identical data types

- Python operators: `+`, `*`, and so on, works for built-in classes
- The same operator works differently with different data types.

In [2]:
```
# add two integers
```

```
2  +  3
```

Out[2]: 5

In [3]:
```
# merge two lists

["1", "2"] + ["3"]
```

Out[3]: ['1', '2', '3']

In [4]:
```
# concatenate two strings

"Python" + "Programming"
```

Out[4]: 'PythonProgramming'

In [5]:
```
# create multiple copies of an element
[0] * 20
```

Out[5]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

In [6]:
```
# duplicate a string
"Python" * 3
```

Out[6]: 'PythonPythonPython'

In [7]:
```
"A" < "a"
```

Out[7]: True

This feature that allows the same operator to have different meaning according to the context is called **operator overloading**.

## Operator overloading

- In OOP, operator overloading is a powerful concept that allows us to define how operators behave for user-defined data types or objects.
- Common operators that can be overloaded include arithmetic operators `(+, -, *, /)` and relational operators `(==, !=, <, >)`
- This means we can define how these operators behave when applied to objects of our classes, making our code more intuitive, readable, and expressive.

In [8]:
```
class Complex:

    # init method to initialise instance variables
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    # string method to display object of Complex class
    def __repr__(self):
        # assuming there is always a real part
```

```
        if self.imag == 0:
            return f"{self.real}"
        elif self.imag > 0:
            return f"{self.real}+{self.imag}i"
        # if imaginary part is negative
        else:
            return f"{self.real}{self.imag}i"
```

In [9]:
```
c1 = Complex(2, 1)
c2 = Complex(5, -7)
```

In [10]:
```
c1
```

Out[10]: 2+1i

In [11]:
```
c2
```

Out[11]: 5-7i

In [12]:
```
c1 + c2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-61818d23e61f> in <module>
----> 1 c1 + c2

TypeError: unsupported operand type(s) for +: 'Complex' and 'Complex'
```

## Arithmetic Operator Overloading

- We cannot perform arithmetic operations on any two objects of a user-defined classs unless we explicitly define a method to handle this within the class.
- To overload arithmetic operators within our class, we use the special methods below:

| Name | Symbol | Special Function |
|---|---|---|
| Addition | + | `__add__(self, other)` |
| Subtraction | - | `__sub__(self, other)` |
| Division | / | `__truediv__(self, other)` |
| Floor division | // | `__floordiv__(self, other)` |
| Modulus | % | `__mod__(self, other)` |
| Power | ** | `__pow__(self, other)` |

See section 3.3.7 in this link for more information on operator overloading in Python.

- Notice that when you add two integers, the result is an integer. Similarly, when you add two strings, the result is a string.
- So, when you implement the `__add__()` method in your class, it is sensible that you return the class instance.

In [13]:
```
class Complex:


    # init method to initialise instance variables
```

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

    # string method to display object of Complex class
    def __repr__(self):
        # assuming there is always a real part
        if self.imag == 0:
            return f"{self.real}"
        elif self.imag > 0:
            return f"{self.real}+{self.imag}i"
        # if imaginary part is negative
        else:
            return f"{self.real}{self.imag}i"

    # ================ Arithmetic Operators ================

    # special method to overload the + operator for our Complex class
    def __add__(self, other):
        return Complex(self.real+other.real, self.imag+other.imag)
```

**Line 22:** Again, the `__add__()` method is used to tell Python what to do when we add two objects created from our `Complex` class.

- `self` is the first parameter of any method defined within the class
- `other` represents the second `Complex` object

**Line 23.** Return value here is `Complex` .

In [14]:
```
c1 = Complex(2, 1)
c2 = Complex(5, -7)
```

In [15]:
```
c1 + c2
```

Out[15]:
```
7-6i
```

In [16]:
```
# can you guess the output here?


c1 - c2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-16-b86d6151af56> in <module>
      1 # can you guess the output here?
      2
----> 3 c1 - c2

TypeError: unsupported operand type(s) for -: 'Complex' and 'Complex'
```

In [17]:
```
class Complex:
```

```
    # init method to initialise instance variables
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    # string method to display object of Complex class
    def __repr__(self):
        # assuming there is always a real part
        if self.imag == 0:
            return f"{self.real}"
        elif self.imag > 0:
            return f"{self.real}+{self.imag}i"
        # if imaginary part is negative
        else:
            return f"{self.real}{self.imag}i"

    # ================ Arithmetic Operators ================

    # special method to overload the + operator for our Complex class
    def __add__(self, other):
        return Complex(self.real+other.real, self.imag+other.imag)

    # special method to overload the - operator for our Complex class
    def __sub__(self, other):
        return Complex(self.real-other.real, self.imag-other.imag)
```

In [18]:
```
c1 = Complex(2, 1)
c2 = Complex(5, -7)
```

In [19]:
```
c1 - c2
```

Out[19]: -3+8i

## Relational Operator Overloading

- Similar to arithmetic operator overloading, we cannot perform relational operations on any two objects of a user-defined classs unless we explicitly define a method to handle this within the class.
- To overload relational operators within our class, we use the special methods below:

| Name | Symbol | Special Function |
|---|---|---|
| Equality | == | __eq__(self, other) |
| Inequality | != | __ne__(self, other) |
| Less than | < | __lt__(self, other) |
| Less than or equal to | <= | __le__(self, other) |

| Name | Symbol | Special Function |
|---|---|---|
| Greater than | > | __gt__(self, other) |
| Greater than or equal to | >= | __ge__(self, other) |

In [20]:
```
a = [1,2,3]
b = [1,2,3]
```

In [21]:
```
# Notice list returns True when we use == to check if
# two lists holds the same elements

print(f"a is b: {a is b}")
print(f"a == b: {a == b}")
```

```
a is b: False
a == b: True
```

In [31]:
```
new_a = a[:]
print(new_a)
new_a is a
```

```
[1, 2, 3]
```

Out[31]:
```
False
```

In [32]:
```
c = "Hello, World!"
d = "Hello, World!"
```

In [33]:
```
print(f"c is d: {c is d}")
print(f"c == d: {c == d}")
```

```
c is d: False
c == d: True
```

In [34]:
```
# Two objects created from the same class Complex with the same variables

c3 = Complex(2, -3)
c4 = Complex(2, -3)
```

In [35]:
```
print(f"c3 is c4: {c3 is c4}")
print(f"c3 == c4: {c3 == c4}")
```

```
c3 is c4: False
c3 == c4: True
```

## The `is` and `==` operator

- `is` Operator:
  - Checks for object identity, i.e., whether two variables reference the same object in memory.
  - Returns `True` if the variables refer to the same object, and `False` otherwise.
- `==` Operator:
  - Checks for equality of values, i.e., whether the content of two variables is the same.

- Returns `True` if the values are equal, and `False` otherwise.

The magic of the equality operator `==` happens in the `__eq__()` method of the object to the left of the `==` sign. If this method is not implemented, then `==` compares the memory addresses of the two objects by default (i.e., it does the same thing as `is`).

In [36]:

```
class Complex:

    # init method to initialise instance variables
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    # string method to display object of Complex class
    def __repr__(self):
        # assuming there is always a real part
        if self.imag == 0:
            return f"{self.real}"
        elif self.imag > 0:
            return f"{self.real}+{self.imag}i"
        # if imaginary part is negative
        else:
            return f"{self.real}{self.imag}i"

    # ================ Arithmetic Operators ================

    # special method to overload the + operator for our Complex class
    def __add__(self, other):
        return Complex(self.real+other.real, self.imag+other.imag)

    # special method to overload the - operator for our Complex class
    def __sub__(self, other):
        return Complex(self.real-other.real, self.imag-other.imag)

    # ================ Relational Operators ================

    # You need to override the equality operator with __eq__
    def __eq__(self, other):
        # first check if other is an instance of Complex
        if isinstance(other, Complex):
            return self.real == other.real and self.imag == other.imag
        # do not compare with objects that are instances of another class
        return "Objects not of the same type"

    # it is sensible to create the inverse of equality (!=)
```

```
    def __ne__(self, other):
        if isinstance(other, Complex):
            return not self.__eq__(other)
        # do not compare with objects that are instances of another class
        return "Objects not of the same type"
```

In [37]:
```
c1 = Complex(2, 1)
c2 = Complex(5, -7)
```

In [38]:
```
# Two objects created from the same class Complex with different
variables

print(f"c1 is c2: {c1 is c2}")
print(f"c1 == c2: {c1 == c2}")
print(f"c1 != c2: {c1 != c2}")
```

```
c1 is c2: False
c1 == c2: False
c1 != c2: True
```

In [39]:
```
# Two objects created from the same class Complex with the same variables

c3 = Complex(2, -3)
c4 = Complex(2, -3)
```

In [40]:
```
print(f"c3 is c4: {c3 is c4}")
print(f"c3 == c4: {c3 == c4}")
print(f"c3 != c4: {c3 != c4}")
```

```
c3 is c4: False
c3 == c4: True
c3 != c4: False
```

In [41]:
```
# try to compare Complex with a list
c5 = [4, -2]
c6 = Complex(4, -2)
```

In [42]:
```
print(f"c5 is c6: {c5 is c6}")
print(f"c5 == c6: {c5 == c6}")
print(f"c5 != c6: {c5 != c6}")
```

```
c5 is c6: False
c5 == c6: Objects not of the same type
c5 != c6: Objects not of the same type
```

**NOTE:**

1. The inverse of equality ( `!=` ) works by default in Python3, as long as `__eq__` is defined.
2. By implementing the `__eq__` method, your class automatically becomes unhashable. Implication is that you cannot store your class in sets and dictionaries.

> An object is *hashable* if it has a hash value which never changes during its lifetime.
>
> For example, a list is *unhashable* because it is mutable, its contents can change at any time.

In [43]:

```
# class Complex is unhashable because of __eq__

new_dict = {c1: "c1"}
#new_set = set([c2])
print(new_dict, new_set)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-43-24333fa7f891> in <module>
      1 # class Complex is unhashable because of __eq__
      2
----> 3 new_dict = {c1: "c1"}
      4 #new_set = set([c2])
      5 print(new_dict, new_set)

TypeError: unhashable type: 'Complex'
```

- If you anticipate that your instance variables will be modified during the lifetime of the object, it is recommended to leave it as unhashable.
- If you are creating an immutable data type, it is recommended that you (re)hash it, whenever `__eq__` is within your class.
- To compare and hash efficiently, use `.__dict__` to access all variables.

In [44]:

```
class Complex:

    # init method to initialise instance variables
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    # string method to display object of Complex class
    def __repr__(self):
        # assuming there is always a real part
        if self.imag == 0:
            return f"{self.real}"
        elif self.imag > 0:
            return f"{self.real}+{self.imag}i"
        # if imaginary part is negative
        else:
            return f"{self.real}{self.imag}i"

    # ================ Arithmetic Operators ================

    # special method to overload the + operator for our Complex class
    def __add__(self, other):
        return Complex(self.real+other.real, self.imag+other.imag)

    # special method to overload the - operator for our Complex class
    def __sub__(self, other):
        return Complex(self.real-other.real, self.imag-other.imag)
```

```
    # ================ Relational Operators ================

    # You need to override the equality operator with __eq__
    def __eq__(self, other):
        # first check if other is an instance of Complex
        if isinstance(other, Complex):
            return self.real == other.real and self.imag == other.imag
        # do not compare with objects that are instances of another class
        return "Objects not of the same type"

    # it is sensible to create the inverse of equality (!=)
    def __ne__(self, other):
        if isinstance(other, Complex):
            return not self.__eq__(other)
        # do not compare with objects that are instances of another class
        return "Objects not of the same type"

    # implement __hash__ to make instances hashable
    # however, all instance variables must be hashable
    def __hash__(self):
        #return hash((self.real, self.imag))
        return hash(tuple(self.__dict__))
```

In [45]:
```
c1 = Complex(2, 1)
c2 = Complex(5, -7)
```

In [46]:
```
# class Student is now hashable because we added __hash__

new_dict = {c1: "c1"}
new_set = set([c2])
print(new_dict)
print(new_set)
```

```
{2+1i: 'c1'}
{5-7i}
```

In [ ]:
```

```

---

# Inheritance

- Inheritance allows us to create new classes (subclasses) based on existing classes (superclasses).
- In Python, you can achieve this by specifying the superclass in parentheses after the new class name.

- This promotes code reusability, improves organization, and simplifies the modeling of hierarchical relationships between objects.

**Key Concepts:**

- **Superclass (Base Class or Parent Class)**: The original class that serves as the foundation for the new class. It defines the core attributes and methods that will be inherited.
- **Subclass (Derived Class or Child Class)**: The new class that inherits properties and functionalities from the superclass. It can add its own unique attributes and methods while still retaining the inherited ones.
- **Inheritance Relationship**: The connection between the subclass and superclass. The subclass "inherits from" the superclass.

In [ ]:
```
import tkinter
import inspect
print(inspect.getsource(tkinter))
```

In [57]:
```
class Student:

    # class variables
    total_students = 0
    all_students = [] # new class variable

    # init method to initialise instance variables
    def __init__(self, first_name, last_name, lab):
        self.first_name = first_name
        self.last_name = last_name
        self.lab = lab
        Student.total_students += 1
        # keep track of each Student object created from our class
        Student.all_students.append(self)

    # instance method to get full name
    def full_name(self):
        return f"{self.first_name} {self.last_name}"

    # instance method
    def mood(self):
        return f"{self.full_name()} enjoys Python programming!"

    # instance method to get email
    def get_email(self):
        full_name = f"{self.first_name.lower()}.{self.last_name.lower()}"
        return f"{full_name}@student.gla.ac.uk"

    # instance method to allow a student change lab
    def change_lab(self, new_lab):
```

```
        self.lab = new_lab

    # class method to get all students
    @classmethod
    def get_all_students(cls):
        # this returns an object representation of each Student created
from our class
        return cls.all_students

     # class method to get students in the same lab
    @classmethod
    def get_students_in_same_lab(cls, lab):
        same_lab = [stud.full_name() for stud in cls.all_students if
stud.lab == lab]
        return same_lab

    # representation method
    def __repr__(self):
        return f"{self.first_name} {self.last_name} -> {self.lab}"
```

In [58]:

```
# New GraduateStudent class inheriting from Student

class GraduateStudent(Student):
    # additional class variable for GraduateStudent
    total_grad_students = 0
    all_grad_students = []

    # additional init method for GraduateStudent
    def __init__(self, first_name, last_name, lab, research_area):
        # calling the init method of the parent class (Student)
        super().__init__(first_name, last_name, lab)
        self.research_area = research_area
        GraduateStudent.total_grad_students += 1
        GraduateStudent.all_grad_students.append(self)

    # additional instance method for GraduateStudent
    def get_research_area(self):
        return self.research_area

    # override the get_email method
    def get_email(self):
        return f"{self.first_name}{self.last_name}@research.uni.ac"

    # override the __repr__ method to include research area
```

```
    def __repr__(self):
        return f"{self.full_name()}. Research area: {self.research_area}"
```

- In the code above, we created a new class called `GraduateStudent`, which inherits from the `Student` class.
- We created new class variables `total_grad_students` and `all_grad_students` specific to the `GraduateStudent` class.
- The `__init__` method of the `GraduateStudent` class calls the `__init__` method of the parent class (`Student`) using `super()`.
- We added a new method `get_research_area` specific to the `GraduateStudent` class
- The `super()` function is used to call the methods of the parent class.
- We overrode the `get_email` and `__repr__` methods from the parent class to include information specific to `GraduateStudent`.

In [59]:
```
student1 = Student("Jacob", "Liu", "LB10")
student2 = Student("Cara", "Lewis", "LB07")
student3 = GraduateStudent("Bonnie", "Shi", "LB02", "Artificial Intelligence")
student4 = GraduateStudent("Kate", "Mykytenko", "LB09", "Data Science")
```

In [60]:
```
student2.get_email()
```

Out[60]:
```
'cara.lewis@student.gla.ac.uk'
```

In [61]:
```
student3.get_email()
```

Out[61]:
```
'BonnieShi@research.uni.ac'
```

In [62]:
```
student3.get_research_area()
```

Out[62]:
```
'Artificial Intelligence'
```

In [63]:
```
student2.get_research_area() # this will raise an error
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-63-c17c3010f04b> in <module>
----> 1 student2.get_research_area() # this will raise an error

AttributeError: 'Student' object has no attribute 'get_research_area'
```

In [65]:
```
student3.total_grad_students # specific to the GraduateStudent class
```

Out[65]:
```
2
```

In [66]:
```
student4.total_students # belongs to the superclass Student
```

Out[66]:
```
4
```

**Key take-aways**:

- Subclasses inherit all attributes and methods from the superclass (except private members).
- Subclasses can override inherited methods to provide their own implementation.
- Use the `super()` function within the subclass to access the superclass's methods and attributes.

In [ ]:

In [ ]:

In [1]:
```
# run this cell to change the width of the current notebook
# this saves you from scrolling to the side when a code line is too long

from IPython.core.display import display, HTML
display(HTML("<style>.container { width:85% !important; }</style>"))
```

In [ ]: