COMP642

Object Oriented Programming

Lectorial 3

Overloading Methods

- The ability of a method to behave in different ways depending on the parameters that are passed to the method.
- It is possible to have several methods of the same name but with different parameters. These methods are referred to as having a different *signature*.

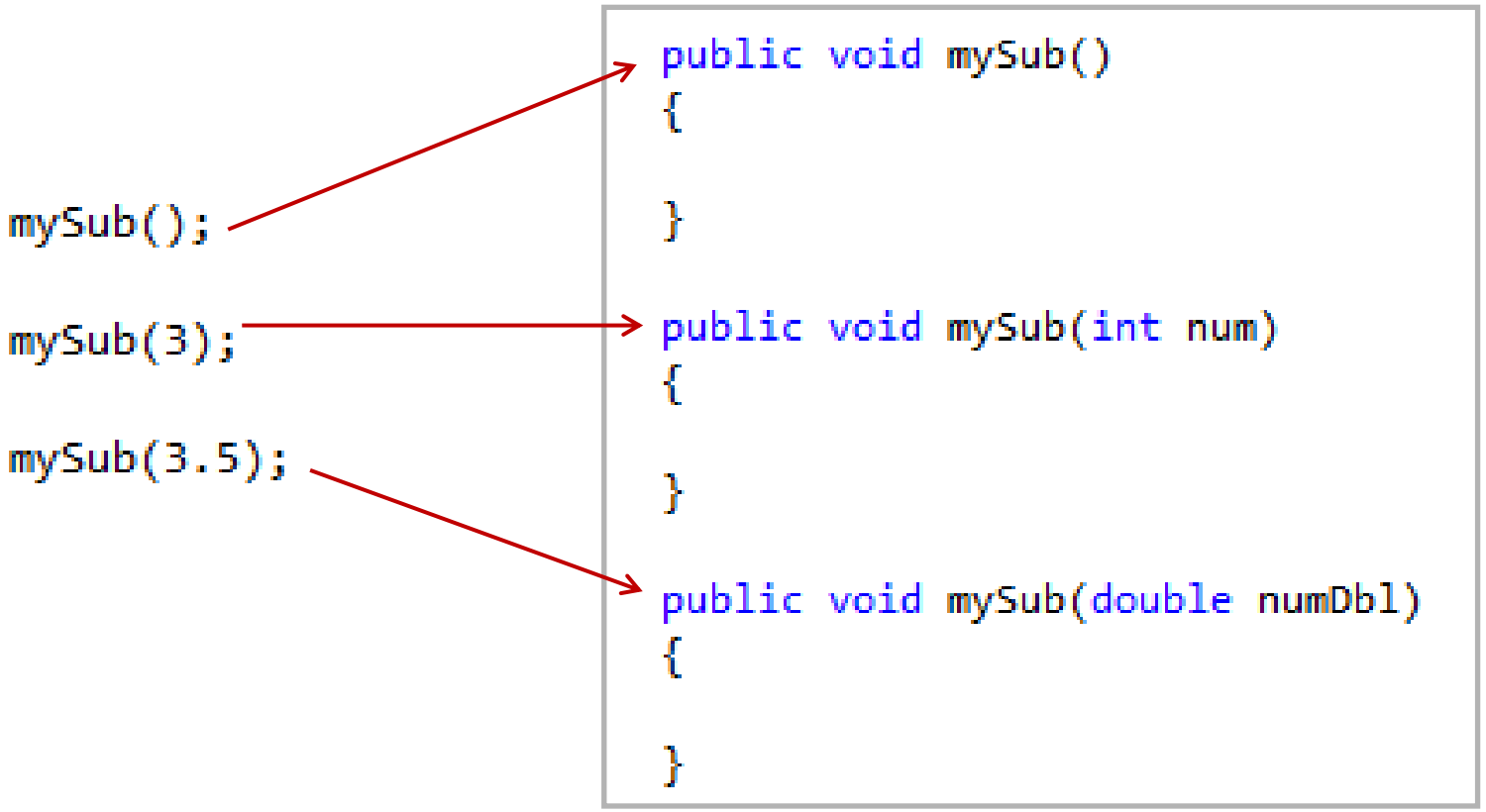
```
public void mySub()
{

}

public void mySub(int num)
{

}

public void mySub(double numDbl)
{

}
```

Same Name

Different Signatures

Calling Overloaded Methods

- When they are called from elsewhere, the appropriate method will be called.



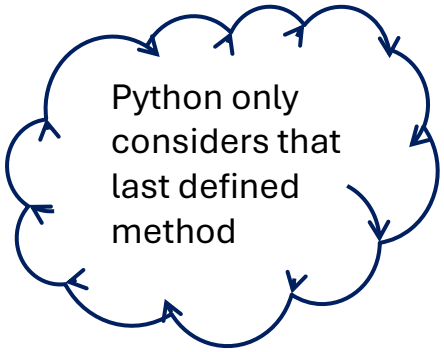
Advantages of Overloading Methods

- Fosters reusability and easy accessibility
- Improves code clarity and eliminates complexity
- Improves the quality of the code
- The method can perform different operations – eliminates the use of different method names for the same kind of operation.

Overloading Methods in Python (1)

- Python does not support method overloading.
- Not possible to define more than one method with the same name in a class.
- Method arguments in Python do not have a type.

```
class Calculate:
    def add(self, a, b):
        print("a + b = {}".format(a + b))

    def add(self, a, b, c):
        print("a + b + c = {}".format(a + b + c))
```




```
calc1 = Calculate()
calc1.add(10, 20, 30)
```

```
a + b + c = 60
```

```
calc1.add(10, 20)
```

```
  File "s:/COMP/COMP642/Sem 2 2021/Python Code/Overloading.py", line 11, in <module>
    calc1.add(10, 20)
TypeError: add() missing 1 required positional argument: 'c'
```

Overloading Methods in Python (2)

- Using default arguments

```
class Calculate:
    def add(self, a, b, c = 0):
        if (c > 0):
            print("a + b + c = {}".format(a + b + c))
        else:
            print("a + b = {}".format(a + b ))
```

```
calc1 = Calculate()
calc1.add(10, 20, 30)
calc1.add(10, 20)
```

```
a + b + c = 60
a + b = 30
```

Overloading Methods in Python (3)

- Using variable length arguments

```python
class Calculate:
    def add(self, *args):
        result = 0
        for param in args:
            result += param
        print("Result: {}".format(result))
```

```python
calc1 = Calculate()
calc1.add(10, 20)
calc1.add(10, 20, 30)
calc1.add(10, 20, 30, 40, 50)
```

```
Result: 30
Result: 60
Result: 150
```

Overloading Built-in Methods in Python (1)

- Operators in Python are overloaded

```
print (10 + 20)
print ("My " + "Name")
```

```
30
My Name
```

```
class Person:
    def __init__(self, firstname, lastname, height):
        self.firstName = firstname
        self.lastName = lastname
        self.height = height
```

```
aPerson = Person("Joe", "Blog", 1.68)
print(aPerson)
```

```
<__main__.Person object at 0x000002043F235E50>
```

Overloading Built-in Methods in Python (2)

```python
class Person:
    def __init__(self, firstname, lastname, height):
        self.firstName = firstname
        self.lastName = lastname
        self.height = height
    def __str__(self):
        return ("Name: {} {} Height: {}".format(self.firstName, self.lastName, self.height))
```

```python
aPerson = Person("Joe", "Blog", 1.68)
print(aPerson)
```



```
Name: Joe Blog Height: 1.68
```

- We can apply overloading to dunder methods such as __add__(), __sub__(), __mult()__, and__len__().

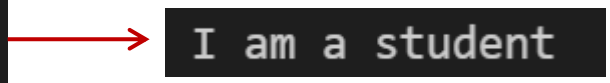
What is a Constructor?

- Special kind of method used to initialise instance members of that class
- When object is created, the constructor definition is the first to execute.
- If there is no constructor, Python will provide a default constructor.

Types of Constructor (1)

- Default Constructor

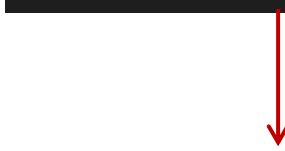
```
class Student:
    def dispMessage(self):
        print("I am a student")

aStudent =Student()
aStudent.dispMessage()
```



Types of Constructor (2)

- Non- Parameterised Constructor

```
class Student:
    def __init__(self):
        print("Student object is created")
    def dispMessage(self):
        print("I am a student")

aStudent =Student()
```



```
Student object is created
```

Types of Constructor (3)

- Parameterised Constructor

```
class Person:
    def __init__(self, firstname, lastname, height):
        self.firstName = firstname
        self.lastName = lastname
        self.height = height
```

```
aPerson = Person("Joe", "Blog", 1.67)
```

Overloading Constructor?

- Constructor cannot be overloaded in Python.
- Can use default argument to get around this.

```
class Person:
    def __init__(self, firstname, lastname = "Unknown", height = 1.50):
        self.firstName = firstname
        self.lastName = lastname
        self.height = height
    def __str__(self):
        return ("Name: {} {} Height: {}".format(self.firstName, self.lastName, self.height))
```

```
aPerson = Person("Joe", "Blog", 1.67)
print(aPerson)

anotherPerson = Person("Mary", "Smith")
print(anotherPerson)

newPerson = Person("Harry")
print(newPerson)
```

```
Name: Joe Blog Height: 1.67
Name: Mary Smith Height: 1.5
Name: Harry Unknown Height: 1.5
```

Encapsulation

- A protective barrier that prevents the code and data being randomly accessed by other code defined outside the class.
- Access to the data and code is tightly controlled by an interface.
- Advantages:
  - no part of the calling program can improperly change the data
  - the class does the checking (don’t need to rely on the programmer of the calling program)
  - changing the methods for setting or retrieving data will not require any changes to the calling code

Public, Protected, Private Members

- Public members are easily accessible from any part of the program.
- All data members and methods of a class are public by default.
- Protected members are only accessible to a class derived from it (More about this later…).
- Private members are accessible within the class only and it is the most secure access modifier.

Public Members

```
class Student:
    def __init__(self, studentName, studentAge):
        self.studentName = studentName
        self.studentAge = studentAge
```

```
aStudent = Student("Harry", 25)
print(aStudent.studentName)
print(aStudent.studentAge)

aStudent.studentName = "Jessica"
print(aStudent.studentName)
```

```
Harry
25
Jessica
```

Can access Student's class attributes and modify the values

Private Members (1)

- Python does not have any mechanism that effectively restricts access to any instance variable or method.
- Python prescribes convention of prefixing the name of the variable/method with a double underscore to emulate the behaviour of private access.
- The double underscore prefixed to a variable makes it private and any attempt to access it outside the class will result in an AttributeError.

Private Members (2)

```
class Student:
    def __init__(self, studentName, studentAge):
        self.__studentName = studentName  #private instance attribute
        self.__studentAge = studentAge      #private instance attribute

    def __display(self):
        print("This is a private method")
```

```
aStudent = Student("Harry", 25)
print(aStudent.__studentName)
print(aStudent.__studentAge)
aStudent.__display()
```

```
    print(aStudent.__studentName)
AttributeError: 'Student' object has no attribute '__studentName'
```

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

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

Private Members (3)

- Python performed name mangling of private variables
- Every member with double underscore will be changed to _object._class__variable.
- It can still be accessed from outside the class

```
aStudent = Student("Harry", 25)
print(aStudent._Student__studentName)
aStudent._Student__display()
```

```
Harry
This is a private method
```

Getter and Setter (1)

- Getters are methods which help access the private attributes or get the value of the private attributes.
- Setters are methods which help change or set the value of private attributes.
- Can be implemented as decorators.

Getter and Setter (2)

- Getter

```
class Student:
    def __init__(self, studentName, studentAge):
        self.__studentName = studentName  #private instance attribute
        self.__studentAge = studentAge    #private instance attribute

    def __display(self):
        print("This is a private method")

    @property
    def studentAge(self):
        print("Getting value..")
        return self.__studentAge
```

```
aStudent = Student("Harry", 25)
print(aStudent.studentAge)
```

→

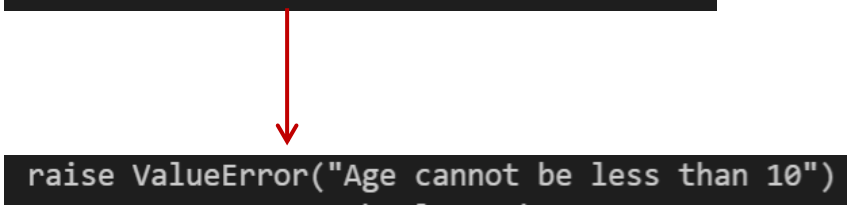
```
Getting value..
25
```

Getter and Setter (3)

- Setter- only accepts valid value

```
@studentAge.setter
def studentAge(self, value):
    print("Setting value..")
    if value < 10:
        raise ValueError("Age cannot be less than 10")
    self.__studentAge = value
```

```
aStudent.studentAge = 5
```



```
raise ValueError("Age cannot be less than 10")
```

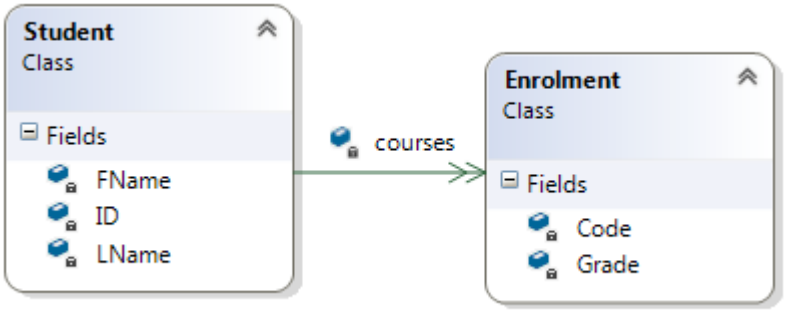
```
aStudent.studentAge = 20
print(aStudent.studentAge)
```

```
Setting value..
Getting value..
20
```

Try This...

We want to keep information about students and their enrolments in subjects.

- What will the classes look like?
- What would be good constructors?



A student has “many” subjects. Keep references to them in a List.

(More about this later...)