Object Oriented Programming

MAD 102

Procedural Programming

- Up to this point we have focused on the concepts often associated with procedural programming
  - We have focused on our programs as a series of steps
  - We execute those steps in sequence
  - We have introduced conditional statements and repetition structures to handle slight deviations in the sequence of our programs
- All the code that we have used has been contained within a single program.

Why Procedural?

- It is easy for us to understand
- It is a good choice for simple programming that we have been implementing
  - Works well with small programs

- Disadvantages
  - It can get out of hand very quickly – it does not scale well
  - It doesn't give us a true idea of what we are working with as attributes (variables) are separated from behaviours (functions)
  - Allows access to data that is uncontrolled and unpredictable since data is global

Object Oriented Programming (OOP)

- Object Oriented Programming is **not a language** but a programming paradigm
  - It is a set of ideas that is supported by other languages
  - It does not replace any existing paradigms, but are an evolutionary response
- OOP contains the **attributes** and **behaviours** within a single object
  - This increases your data integrity
- OOP also allows for **reusability** and **extensibility**

Reusability

- You will often find that you are using code that you have used before – think about the addition, subtraction, multiplication, division code that you created and re-used in other assignments
- OOP allows for this easy re-use which:
  - Saves time
  - Improved reliability – you are using something that you already know works!

Extensibility

- Program are used for a long time
- We can extend the life of our program by making small updates to our existing code
- OOP makes it easier to extend or add on to the existing code

What is an object?

- Objects are all kinds of *things*
- Some of these things are tangible – I can show it to you, touch it
  - Person
  - Car
  - Plant
  - Animal
- Some of these things are intangible – they exist, but I can't present it to you to touch
  - Bank Account

What is an object?

- In many cases, the objects are nouns.
- For example – think about a system for registering students:
  - The *student* will need a student number
  - The *student* will need to choose a course
  - The *student* will need to know course(s) they are enrolled in
  - **NOTE – if you can put the word the in front of it, it is generally considered to be a noun – *the* student, *the* cup, *the* car…**
- The ***student*** would be a great example of an object

Objects

- All objects share some common characteristics:
  - Identity – we need to know how to identify one object from the next
    - Two people may be similar, but we have a way of identifying both of them
  - **Attributes** - things about them
    - People have names, heights, birth dates, etc.
  - **Behaviours** – things you can do
    - People can talk, run, jump, calculate the square root of 144

Objects

- Objects are self contained
  - Each object has its own identity, attributes and behaviours
  - Objects can have the same attributes and behaviours – but we use identity to distinguish one from the other.

Attributes

- **Attributes** represent data that is stored within an object
- This data represents the **state** of the object
- The values can help distinguish one object from the rest

If we were to describe a town - what might we use to describe one from the other?

Attributes

- We represent attributes (often called **properties**) of an object using variable values
- If we took a look at a town, it may have:
  - A property for the town name
  - A property for the town's population
  - A property for the number of stop lights it has
  - A property to tell if it has a Starbucks or not

Behaviours

- These are represented by what are known as **methods**
  - They are functions that the object can perform
- If we take a look at a dog – what might some actions that it could perform
  - It could bark
  - It could eat
  - It could run
  - It could sleep

Abstraction

- Abstraction is an OOP concept
- It is a process where you only show the data that is relevant to the user
- Any other information is hidden away
- Think of logging into a site
  - We enter a name and a password and then press the login button
    - How the input is verified, sent, received is all abstracted away from us
- Object do not need to be fully and completely defined – just the essential information

Encapsulation

- Think of encapsulation as wrapping your object in a box
- You are encapsulating the functionality and only allowing someone or something to interact with what you want them to
  - Hiding away any other functionality
- A single object contains all of its data and behavior but only reveals what it wants to
  - Restricting access to an objects internal mechanism helps ensure that change can be controlled

Inheritance

- Inheritance is a concept that allows for the code reuse and better overall design
- Inheritance allows an object to ***inherit*** (*share*) attributes and methods from another object
  - Allows for the organization of code by factoring in commonalities
  - Think about an animal object – it could have a colour of eyes property and a walk method.
  - We could also have a dog object that is not only an animal object (it will share the colour of eyes property and walk method) but it also has a bark method
  - And we could have a cat object – that is an animal object (it will share the colour of eyes property and walk method) but it has a meow method

Polymorphism

- Is a Greek word meaning “many shapes” or “many forms”
- It is an idea that allows you to process objects differently
- For example – if we had an object that was a Vehicle and that vehicle had a method for making noise.
- If we make an object called car that was a vehicle and it inherited the ability to make a noise, it might make the noise ”HonkHonk”
- We could also have an object called train that was also a vehicle and it inherits the ability to make a sound but instead it says “Choo choo”
- Two objects that have different implementations – but are still derived from the same parent object

OOP

- OOP can be a tough concept to grasp
- It is important that you understand it – and **practice, practice, practice**

Creating a class

- The **class** keyword is used to create a user-defined type of object containing groups of related attributes (**properties**) and actions (**methods**)
- Classes are given names (usually singular) that describe them
  - Class names begin with an **uppercase letter**
  - For example – we wanted to model some students we could create a class called **`Student`**

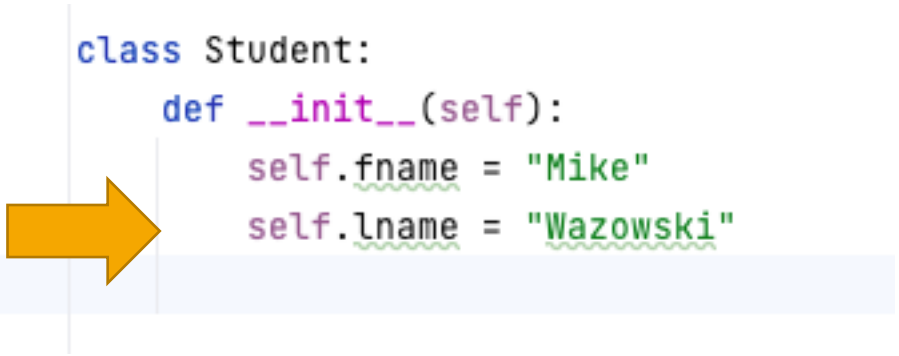
Creating a class

- Classes have attributes that describe them
  - This data is called **properties**
- When an instance of a class is created, a process called **instantiation** is completed
- All properties of the class have values set by a method on the class commonly called a **constructor**

Constructor

- The constructor sets up the initial state of the new instance
  - This means setting the initial values for the **instance's properties**
  - The method is the `__init__`
  - The format follows that of a function `def __init__ (self):`

Instance properties



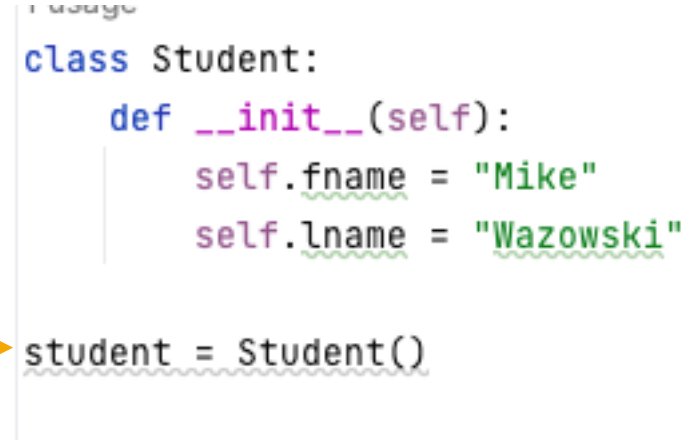
self

- The keyword **self** is a method parameter that refers to the class instance
  - It is used in the init method and elsewhere in the class body to refer to the specific instance

Creating instances

- With a constructor in place, you can create a new instance of the student
- Assign the class to a variable – the variable will represent the new student

New instance

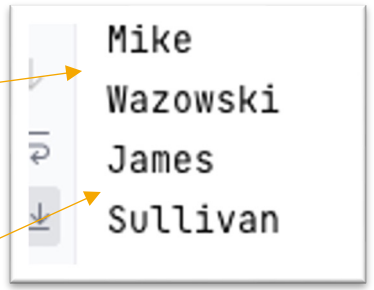


Creating new instances

```python
class Student:
    def __init__(self):
        self.fname = "Mike"
        self.lname = "Wazowski"


student = Student()
print(student.fname)
print(student.lname)

another_student = Student()
another_student.fname = 'James'
another_student.lname = 'Sullivan'
print(another_student.fname)
print(another_student.lname)
```



Accessing property values

- The values of the instance can be accessed using the attribute reference operator (member operator)
  - This is dot notation - the instance followed by a . followed by the property

```
3   class Student:
4       def __init__(self):
5           self.fname = "Mike"
6           self.lname = "Wazowski"
7
8   student = Student()
9   print(student.fname)
10  print(student.lname)
11
```

```
Mike
Wazowski
```

Modifying state

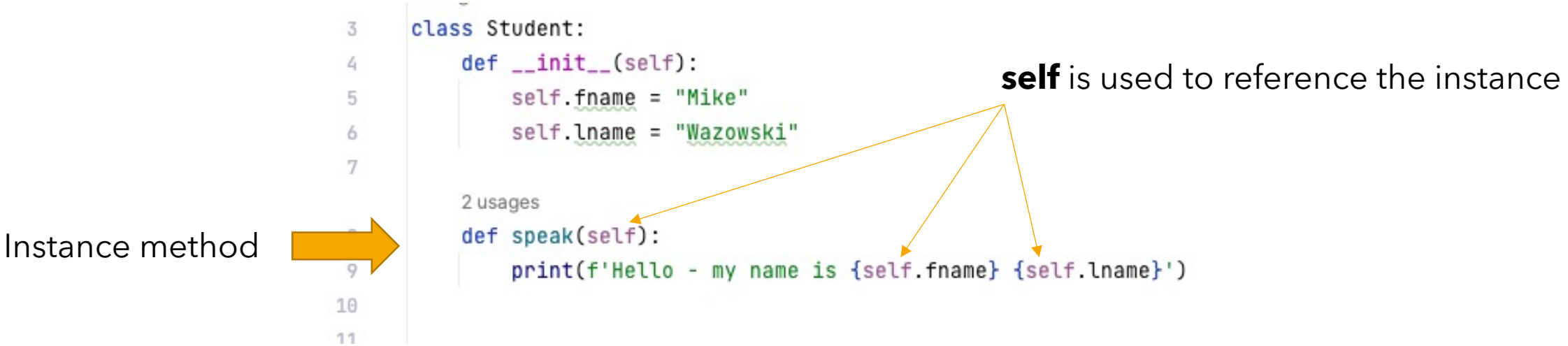
- The state of your instance can be modified using the same dot notation

```
class Student:
    def __init__(self):
        self.fname = "Mike"
        self.lname = "Wazowski"

student = Student()
print(student.fname)
student.fname = 'Bob'
print(student.fname)
```

Methods

- Methods are actions that your classes can perform
  - They are functions defined within the class
  - Known as instance methods – **actions** that are **called** on the **instance**



Methods

- To access the instance methods – use dot notation

```
2 usages
class Student:
    def __init__(self):
        self.fname = "Mike"
        self.lname = "Wazowski"

    2 usages
    def speak(self):
        print(f'Hello - my name is {self.fname} {self.lname}')


student = Student()
student.speak()

another_student = Student()
another_student.fname = 'James'
another_student.lname = 'Sullivan'
another_student.speak()
```

NOTE- self is not included when the method is called

```
Hello - my name is Mike Wazowski
Hello - my name is James Sullivan
```

Class Object/ Instance Object

- A class object acts as a factory that creates instance objects
- An instance object is created by the init method

Class Object

Instance Object

```
class Student:
    def __init__(self):
        self.fname = "Mike"
        self.lname = "Wazowski"

    2 usages
    def speak(self):
        print(f'Hello - my name is {self.fname} {self.lname}')


student = Student()
student.speak()
```

Class Attribute

- A class attribute is an attribute that is shared by all instance of the class
  - They are defined within the scope of the class
  - They are still accessed on the instance
  - They are modified on the instance or the class
- Class attributes are for details that all instances of the class share
- Instance attributes are for data that is unique to each instance

Class Attribute

```
2 usages
class Student:
    school = 'St Clair College'

    def __init__(self):
        self.fname = "Mike"
        self.lname = "Wazowski"

    2 usages
    def speak(self):
        print(f'Hello - my name is {self.fname} {self.lname}')


student = Student()
student.speak()
print(student.school)

another_student = Student()
another_student.fname = 'James'
another_student.lname = 'Sullivan'
another_student.speak()
print(student.school)

```

Class attribute – all students will share the same school

Instance attributes – all students will have different first and last names – part of the init

```
Hello - my name is Mike Wazowski
St Clair College
Hello - my name is James Sullivan
St Clair College
```

Class Attribute

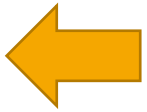
```
3   class Student:
4       school = 'St Clair College'
5
6       def __init__(self):
7           self.fname = "Mike"
8           self.lname = "Wazowski"
9
        2 usages
10      def speak(self):
11          print(f'Hello - my name is {self.fname} {self.lname}')
12
13
14  student = Student()
15  student.speak()
16  print(student.school)
17
18  another_student = Student()
19  another_student.fname = 'James'
20  another_student.lname = 'Sullivan'
21  another_student.speak()
22  print(student.school)
23  Student.school = 'Monsters University'
24  print(student.school)
25  print(another_student.school)
```

```
Hello - my name is Mike Wazowski
St Clair College
Hello - my name is James Sullivan
St Clair College
Monsters University
Monsters University
```

Class constructors

- In the previous examples, we are creating new instances that have the same state – and then we modify the attribute values to create the new current state of the object

```
3   class Student:
4       def __init__(self):
5           self.fname = "Mike"
6           self.lname = "Wazowski"
7
        2 usages
8       def speak(self):
9           print(f'Hello - my name is {self.fname} {self.lname}')
10
11
12  student = Student()
13  student.speak()
14
15  another_student = Student()
16  another_student.fname = 'James'
17  another_student.lname = 'Sullivan'
18  another_student.speak()
```

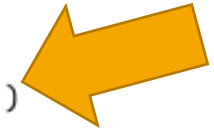


Class constructors

- Simplify the process to allow for the customization of the instance
  - Provide a list of parameters for values
- Arguments passed into the constructor are the values used to initialize the instance

Class constructor

```
3   class Student:
4       def __init__(self, fname, lname):
5           self.fname = fname
6           self.lname = lname
7
        2 usages
8       def speak(self):
9           print(f'Hello - my name is {self.fname} {self.lname}')
10
11
12  student = Student('Mike', 'Wazowski')
13  student.speak()
14
15  another_student = Student('James', 'Sullivan')
16  another_student.speak()
17
```



```
Hello - my name is Mike Wazowski
Hello - my name is James Sullivan
```

Default Values

- Default parameters can be used for what might be common for all instances, but may be unique in others
  - If it is common, use a class attribute
- If a value is provided as part of the initialization process, the passed value will be used
- If no value is provided as part of the initialization process, the default value will be used

Default Values

```python
class Student:
    def __init__(self, fname, lname, program="Programming"):
        self.fname = fname
        self.lname = lname
        self.program = program

    2 usages
    def speak(self):
        print(f'Hello - my name is {self.fname} {self.lname} - enrolled in {self.program}')


student = Student('Mike', 'Wazowski')
student.speak()

another_student = Student('James', 'Sullivan', 'Engineering')
another_student.speak()
```

Used default

Used custom

```
Hello - my name is Mike Wazowski - enrolled in Programming
Hello - my name is James Sullivan - enrolled in Engineering
```

Class interfaces

- A class interface consists of the methods that a programmer calls to create, modify or access a class instance
  - The class may contain methods that access or modify the attributes of the class - but you may want to restrict access to those methods
  - Good practice is to prepend an underscore to the method to demonstrate that the method should only be used internally by a class

Class interfaces

- Python lacks the ability to hide or restrict access to its internal code
  - Other languages use keywords like public or private which restrict access to variables or methods

Memory allocation

- Memory allocation is the process of an application requesting and being granted memory
  - Some languages require you to write memory allocating code, Python runtime handles this for you

Memory Deallocation

- When an object is no longer required, the memory can be freed
  - This process is called memory deallocation
- This occurs when an object is no longer referenced by any variables
  - The number of variables referencing an object is its reference count – an integer value
  - When the reference count reaches 0 – Python's garbage collector will deallocate the object