bianca.phelan@tudublin.ie

O Object
O Oriented
P Programming

S1 = Python with Bianca

S2 = Java with Bryan

DT228(TU856)/DT282(TU858) - 2

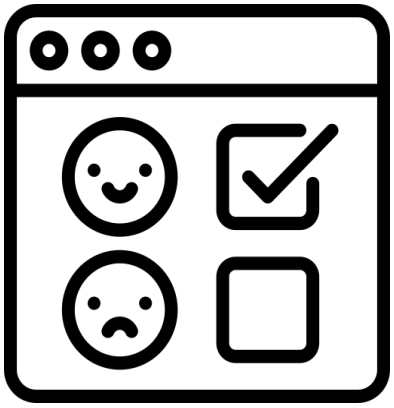
OLLSCOIL TEICNEOLAÍOCHTA BHAILE ÁTHA CLIATH
TU DUBLIN
TECHNOLOGICAL UNIVERSITY DUBLIN

COMPUTER SCIENCE

Objectives

- Discuss survey result
- Experiment with static vs class methods in Python
- Analyse the principle of abstract classes
- Program Polymorphism in Python

Some Feedback Points from the Survey



Icon from flaticon.com

- One directory for all lecture slides?
  - Brightspace organisation
- Too many new languages this year!
- Revision on git
- A main template for writing something in Python
- Debugger?!
- Exception handling
- Labs are too big, I want smaller question based exercises
  - 1ECTS = 20-25hrs student input

Static Methods and Class Methods

Different Methods

- **Instance** method
  - What we have seen so far
  - Attached to the instance of a class via `self`
  - Has access to instance variables and class variables
- **Class** method
  - Attached to the class via `cls`
  - Can only access class attributes, not instance attributes
- **Static** method
  - Not attached to the class
  - Is grouped inside this class because there is some logical connection between the class and the method
  - Looks like a class-less function
  - No access to instance variables

Remember that `self` is only a naming convention. `cls` is the same.

Different Methods

```
class DifferentMethodsClass:
    class_attribute = "This is a class attribute"

    def __init__(self):
        self.instance_attributes = "This is an instance attribute"

    def instance_method(self):  # usual argument self
        print('instance method called', self)

    @classmethod
    def class_method(cls):  # notice what's new in the argument list
        print('class method called', cls)
        # print(self.instance_attributes)  # this will fail
        print("class attribute: ", cls.class_attribute)

    @staticmethod
    def static_method():  # notice nothing in the argument list
        print('static method called')
```

Different Methods

```
class DifferentMethodsClass:
    class_attribute = "This is a class attribute"

    def __init__(self):
        self.instance_attributes = "This is an instance attribute"

    def instance_method(self):  # usual argument self
        print('instance method called', self)

    @classmethod
    def class_method(cls):  # notice what's new in the argument list
        print('class method called', cls)
        # print(self.instance_attributes)  # this will fail
        print("class attribute: ", cls.class_attribute)

    @staticmethod
    def static_method():  # notice nothing in the argument list
        print('static method called')
```

```
demo = DifferentMethodsClass()
# all of these work just ine
demo.instance_method()
demo.class_method()  # does not have access to instance variables
demo.static_method()
```

```
DifferentMethodsClass.class_method()  # works fine
DifferentMethodsClass.static_method()  # works fine
# # DifferentMethodsClass.instance_method()  # error
```

Example of a Static Method

```
pizza = Pizza(['cheese', 'tomatoes'])
print(pizza)
print(Pizza.prosciutto())
print(Pizza.margherita())

# print(Pizza.ingredients) # causes an error, cannot access instance variables
print(Pizza.pizza_area(3))
```

```
import math
class Pizza:
    def __init__(self, ingredients, pizza_size=3):
        self.ingredients = ingredients
        self.pizza_size = pizza_size
        print("in init")

    def __str__(self):
        return f'Pizza({self.ingredients}) of size: {self.pizza_size}'

    # add some class methods
    @classmethod
    def margherita(cls):
        return cls(['mozzarella', 'tomatoes'])

    @classmethod
    def prosciutto(cls):
        return cls(['mozzarella', 'tomatoes', 'ham'])

    @staticmethod
    def pizza_area(r):
        return r ** 2 * math.pi
```

[1]

Abstract Classes And Polymorphism

Abstract Classes

Abstract Classes

- A class that does not have any implementations by itself, but provides the structure of a class using methods
- All classes inheriting from this abstract class must implement these methods

Why Use Abstract Classes

- To provide a common application program interface (API) for a set of subclasses
- Especially useful if a third party is going to produce the functionality, maybe via a plugin
- Also useful for very large projects

Abstract Base Class

- Abstract Base Class, short **`ABC`**
- Define a set of methods and properties that a class that inherits from it must implement in order to be considered a duck type instance of that class
- Usually you won't need to create a new ABC as many are provided by the Python standard library

Comparison

Not a basic concept in Python. Abstract classes are implemented via the Abstract Base Class module.

Abstract Classes

- Abstract class should not be instantiated, but in Python we are allowed to
- Use ABC module
- Design question
- Used to generate new types
- Extract core features without dealing with actual implementation

Inheritance

- Inheriting behaviours and states but having some of your own as well
- All behaviours are implemented

Example Abstract Classes

- Define an abstract base class called `SuperHero` with methods like `suit_up()` and `rescue_world()` but provides **no** implementation for these methods, they are specific to the individual super hero. You would also not create an object of `SuperHero` but only of a specific hero
- `BatMan` and `SuperMan` both inherit from `SuperHero` and implements what `suit_up()` and `rescue_world()` actually do
- We only need to override methods that we **actually need**, for example some super heros might not need to suit up
  - In Python needs to be mentioned anyway but you can choose to just add pass

Real Abstract Class vs Not a Real Abstract Class

With or without ABC, only real abstract with decorator

Not a real Abstract Class

```
class Polygon(ABC):

    # abstract method
    def no_of_sides(self):
        pass


class Triangle(Polygon):
    pass
```

Real Abstract Class

```
class Polygon(ABC):

    @abstractmethod
    def no_of_sides(self):
        pass


class Triangle(Polygon):
    # overriding abstract method
    def no_of_sides(self):
        print("I have 3 sides")
```

Notice the use of the decorator. It forces an implementation in the derived class.

ABC with Implementation

- A class derived from an abstract class cannot be instantiated unless the abstract methods have been overridden
- Abstract methods in the base class can contain implementation logic
  - Can be accessed via super()
  - Will still force an overriding of the method
- `super()` works just as before
- Sometimes we see examples online inheriting from `metaclass=ABCMeta`
  - That's what's under the hood of inheriting from ABC
  - Metaclass next slide

What is a meta class?

- Not all OOP languages support this concept, Python does
  - Those programming languages that support meta classes each do it in their own way
- A meta class is a class whose instances are classes
- Use cases:
  - Logging and profiling
  - Interface checking
  - Registering classes at creation time
  - Automatic property generation,
  - Automatic resource locking, synchronisation tasks, etc

More Examples on Abstract Class

```
class Polygon(ABC):

    @abstractmethod
    def no_of_sides(self):
        pass


class Triangle(Polygon):
    # overriding abstract method
    def no_of_sides(self):
        print("I have 3 sides")
```

```
t = Triangle()
t.no_of_sides()
```

```
class Pentagon(Polygon):

    # overriding abstract method
    def no_of_sides(self):
        print("I have 5 sides")
    # pass  #causes an error. Als

p = Pentagon()
p.no_of_sides()
```

```
/Users/bianca.scho
I have 3 sides
I have 5 sides
```

Example Abstract Class

```
class Polygon(ABC):

    @abstractmethod
    def no_of_sides(self):
        pass

    def what_am_I(self):
        print("I am a parent Polygon")


class Triangle(Polygon):
    # overriding abstract method
    def no_of_sides(self):
        print("I have 3 sides")


t = Triangle()
t.no_of_sides()
t.what_am_I()
```

```
/Users/bianca.schoenphe
I have 3 sides
I am a parent Polygon
```

The derived class has access to all methods of the base class.

[5]

Example Abstract Class: Call to super()

```
class Polygon(ABC):

    @abstractmethod
    def no_of_sides(self):
        pass

    def what_am_I(self):
        print("I am a parent Polygon")


class Triangle(Polygon):
    # overriding abstract method
    def no_of_sides(self):
        print("I have 3 sides")

    # pass

    def what_am_I(self):
        print("I am a Triangle child class")
        super().what_am_I()
```

```
t = Triangle()
t.no_of_sides()
t.what_am_I()  # everything works as expected
```

```
/Users/bianca.schoenphe
I have 3 sides
I am a parent Polygon
```

Calls to super() work as before.

[5]

Example Abstract Class: Property

```python
class Polygon(ABC):

    @abstractmethod
    def no_of_sides(self):
        pass

    def what_am_I(self):
        print("I am a parent Polygon")

    @property
    @abstractmethod # the abstract deco
    def length(self):
        pass
```

Property should go first, abstractmethod last.

```python
class Triangle(Polygon):

    def __init__(self):
        self.__x = 0

    def no_of_sides(self):
        print("I have 3 sides")

    def what_am_I(self):
        print("I am a Triangle child class")
        super().what_am_I()

    @property
    def length(self):
        return self.__x

    @length.setter
    def length(self, value):
        self.__x = value
```

Example Abstract Class: Property

```
class Polygon(ABC):

    @abstractmethod
    def no_of_sides(self):
        pass

    def what_am_I(self):
        print("I am a parent Polygon")

    @property
    @abstractmethod # the abstract deco
    def length(self):
        pass
```

```
    def __init__(self):
        self.__x = 0

    def no_of_sides(self):
        print("I have 3 sides")

    def what_am_I(self):
        print("I am a Triangle child class")
        super().what_am_I()

    @property
    def length(self):
        return self.__x

    @length.setter
    def length(self, value):
        self.__x = value
```

```
t = Triangle()
t.no_of_sides()
print(t.length)
t.length = 5
print(t.length)
```

```
/Users/bianca.s
I have 3 sides
0
5
```

Polymorphism

Polymorphism

- The same function is defined for objects of different types
- Different behaviours happen depending on which subclass is used without having to explicitly know which subclass to use
- Example:
  - `audio_file.play_music()`
  - The action behind this statement will be very different depending the type of audio file, as some formats are compressed and others are uncompressed and those that are compressed use different compression algorithms

`len(...)` is an in-built polymorphic function in Python; another example: `+` operator.

Polymorphism cont'd

- Each type of file can be represented by a different class
- Base class AudioFile and WavFile or MP3File inherit from AudioFile,
- All have a play_music() method that works according to that file format's requirements
- A MediaPlayer object wouldn't need to know which object to use, just calls play_music()
- All subclasses check that the right extension is given at the start

Example 1 Polymorphism

```
class AudioFile:
    def __init__(self, filename):
        if not filename.endswith(self.ext):
            raise Exception("Invalid file format")

        self.filename = filename

class MP3File(AudioFile):
    ext = "mp3"
    def playMusic(self):
        print("playing {} as mp3".format(self.filename))

class WavFile(AudioFile):
    ext = "wav"
    def playMusic(self):
        print("playing {} as wav".format(self.filename))
```

[4]

Example 2 Polymorphism

```
class Cat:
    def makeSound(self):
        print('meow')


class Dog:
    def makeSound(self):
        print('wouff wouff')


def speak(animalType):
    animalType.makeSound()
```

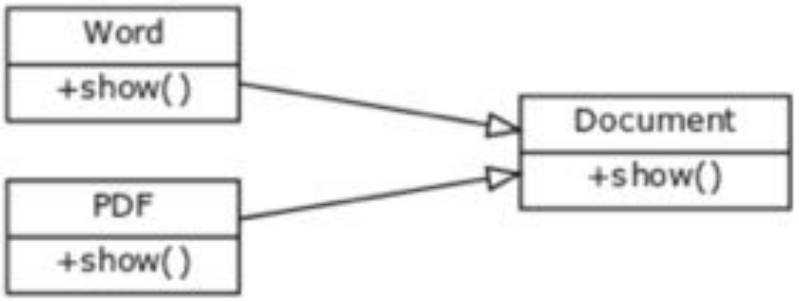
```
cat = Cat()
dog = Dog()

speak(cat)
speak(dog)
```

```
AnimalsPoly
/Users/bianca
meow
wouff wouff
```

[8]

Example 3 Polymorphism




```
for document in documents:
    print document.name + ': ' + document.show()
```

- Imagine you are developing an editor, but you don’t know yet which file types it should be able to open
- Instead of having different types, just access them through Document

Summary

- ★ **Class method vs Static method**
- ★ **Abstract Classes**
- ★ **Polymorphism**

References

1. Static and class methods, Real Python, https://realpython.com/instance-class-and-static-methods-demystified/, accessed Nov 2020
2. Python 3: Object-oriented programming, 2nd edition, Dusty Phillips, 2015, Packt Publishing.
3. Python Course, Abstract Classes, https://www.python-course.eu/python3_abstract_classes.php, accessed Oct 2018.
4. Python Tutorials, Polymorphism, https://pythonspot.com/polymorphism/, accessed Oct 2018.
5. Python Abstract Class Example, geeks for geeks, # *https://www.geeksforgeeks.org/abstract-classes-in-python/#:~:text=An%20abstract%20class%20can%20be,is%20called%20an%20abstract%20class*, accessed Nov 2020.