Object Oriented Programming

Overview

- Introduction to Object Oriented Programming (OOP)
- Built-in Types
- Defining our own Types
- The Four Principles of OOP

Learning Objectives

- Define Object Oriented Programming
- Understand Types, Classes and Objects
- Discover how we can use Type-Hinting to make our code cleaner
- Identify the four principles of OOP
- Implement OOP using Python

What is OOP?

A programming paradigm based on the concept of "objects", which can contain data and code.

Data is in the form of fields, often known as attributes or properties.

Code is in the form of procedures, often known as methods.

Most popular programming languages are class-based, meaning that "objects" are instances of *classes*, which also determines their *type*.

Definitions

- **Type**: What something is a type of, such as a string, integer, boolean, etc.
- **Class**: A blueprint which defines the properties and methods of a *type*. Describes what makes a string type different from an integer type, for example.
- **Object**: An instance of a *type* which is modelled on a *class*. A specific string or integer value, for example.

Built-in Types

The *class* definitions that describe the primitive types (String, Number, Boolean etc...) are built into Python and this complexity is **abstracted** away from us. This helps us to write programs quicker and is part of what makes Python a high-level language.

A primitive type is used to build more complex types.

In a low-level language, we'd have to create all our types manually.

Example

When we create a new string, we actually instantiate an object of *type* `str`. Each string is made from the same *class* so it has the same methods and properties available to it.

```
msg = "hello, I'm a string value!" # is an object of type `str`

# `msg` has access to the `str` methods defined by the `str` class

msg.count() # 24
msg.capitalize() # Hello I'm a string value
msg.upper() # HELLO I'M A STRING VALUE
#...
```

What do we think so far?

- Built-in types make our life a lot easier
- Without the string methods, we'd have to create that functionality ourselves
- With everything being an object, handy bits of functionality are attached to the thing they represent

Start a discussion around these points and ask them for thoughts. The idea here is to check understanding, and reveal some of the OOP principles coming next.

Defining Our Own Types

Imagine trying to keep track of a number of people and their attributes. We could represent each person as an object of *type* `list`:

```
#         Name        Age?  Height? Weight?
jane = ['Jane Doe', 20,  175,    150]
john = ['John Doe', 30,  180,    160]
joe =  ['Joe Doe',  40,  190,    170]
```

What if we're managing 100s or 1000s of people like this? What are some problems that might occur?

Discuss with the learners why maintaining people like this is problematic.

1. What if we want to add/remove attributes?
2. How do we know which value represents which attribute?
3. How do we enforce required / optional attributes?
4. What about default values?
5. Is there any specific logic we would want to perform on a person?

Say that a great way to manage code like this is to create a custom `Person` *type*.

Defining a Person Type

A *type* definition begins with the `class` keyword.

```
class Person:
    #...
```

The `class` is a template for the object. We can `initialise` an `instance` of a class from this template like so:

```
jane = Person()
```

Note: Custom *types* follow the `PascalCase` naming convention.

PascalCase - popularised by the Pascal programming language camelCase kebab-case snake_case - name origin unclear, but kinda looks like snake, stays low down to the ground

---

The Constructor

The properties that form our initial `Person` are defined in a method called `__init__`.

This is commonly referred to as the **constructor** as it constructs the initial state of our object.

```
class Person:
  # Constructor
  def __init__(self, name, age):
    # Properties
    self.name = name
    self.age = age
    print("Created new person")
```

That is, __init__() initialises each new instance of the class.

You can give it any number of parameters but the first is always self.

We can also define default properties(props) here too.

Calling The Constructor

The constructor is called when an `instance` of that class is `initialised` like `jane = Person("Jane", 26)`.

```
class Person:
    # Constructor
```

```
    def __init__(self, name, age):
        # Properties
        self.name = name
        self.age = age
        print("Created new person")

jane = Person("Jane", 26) # Initialising a new person
```

This program prints the words `Created new person` because the `__init__` function is called when `initialising` a new person.

Self

`self` is a **reference** to the current instance of the class.

It is used to access variables and methods that belong to itself.

It is always the first argument when defining methods.

```
class Person:
  # Constructor
  def __init__(self, name, age):
    self.name = name
    self.age = age
```

It doesn't have to be named self, but it's typical practice.

Self is not a keyword either.

Adding Methods

```
class Person:
  # Constructor
  def __init__(self, name, age):
    # Properties
    self.name = name
    self.age = age

  # Method
  def increment_age(self):
    self.age += 1
```

Creating a Person

The built-in `int` type can be constructed from a string representation `int("1234")`.

Our custom `Person` type can similarly be constructed by passing the required values into the *constructor* of the Person class.

```
john = Person('John', 10) # is an object of type `Person`

# Just like `str` we can access its
# properties(props) and methods using dot notation

john.age # 10
john.increment_age()
john.age # 11
```

A note on object types

```
john                # is an object of type `Person`

john.age            # is an object of type `int`

john.name           # is an object of type `str`

john.name.upper()   # we still have access to all the string methods
```

Quiz Time! 🤓

What will this code output?

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def increment_age(self):
        self.age += 1

    def introduce_yourself(self):
        print(f"Hello my name is {self.name}, I am {self.age} years old.")

p = Person("Alice", 25)
p.increment_age()
p.introduce_yourself()
```

1. Hello my name is Alice, I am 25 years old.
2. Hello my name is Alice, I am 26 years old.

3. Hello my name is 25, I am Alice years old.
4. NameError: name 'self' is not defined.

Answer is `2. Hello my name is Alice, I am 26 years old.` because the age is incremented before introduce_yourself is called.

Answer: 2

Type Hinting

- When defining a function, you can define what `type` the arguments are.
- The type for each argument is defined after the colon `a: int`.
- The return type of the function is shown by the arrow `-> int`.

For example `add_numbers` takes two integers and `add_strings` takes two strings.

```
# Arguments a and b are of type int
def add_numbers(a: int, b: int) -> int: # This function return an int
    return a + b

# Arguments a and b are of type string
def add_strings(a: str, b: str) -> str: # This function returns a str
    return a + b

add_numbers(1, 2)
add_strings("hello", " world")
```

Type hinting does not actually enforce at runtime the data that is passed; type hinting is only to help the developer.

Some development tools may also warn you about it if configured to do so. Can get type hinting validation in VSCode with Pylance extension: https://www.emmanuelgautier.com/blog/enable-vscode-python-type-checking/

Benefits of Type Hinting

By defining the types in the arguments, VSCode will suggest the relevant functions for that type. For example a `str` will suggest the `title` and `capitalize` functions.

```
def add_strings(a: str, b: str):
    return a.title() + b.capitalize()
```

This allows us as developers to better express the intent of our code, and to work with code ourselves and other developers have written without ambiguity.

Quiz Time! 🤓

Which of the following is the correct usage of type hinting?

```
def add_numbers(a, b) -> int: #1

def add_numbers(a int, b int) --> int: #2

def add_numbers(a: int, b: int) --> int: #3

def add_numbers(a: int, b: int) -> int: #4
```

Answer: 4

Type Hinting for Complex Types

We can use any Python types in the typing module. For example, in `print_cars` we are using `List[Dict[str, str]]` to show the argument is a list of dictionaries.

```
from typing import List, Dict  # Import the List and Dictionary types

def print_cars(cars: List[Dict[str, str]]):  # The cars argument is a list
of dictionaries
    for car in cars:
        print(f"Brand: {car['brand']}, Colour: {car['colour']}")

# This cars variable is type List[Dict[str, str]]
cars = [{"brand": "BMW", "colour": "Blue"},
        {"brand": "Volvo", "colour": "Red"}]

print_cars(cars)
```

Type Hinting with our own Classes

Once you've defined your own types, you can use them just like built in types.

```
class Person():
    name = "Jane"
    age = 26


# This function takes an argument person which is of type Person
def greet_person(person: Person):
    print(f"Hello {person.name}")
```

```
jane = Person()

greet_person(jane)
```

Complex Type Hinting with Classes

We can combine our types with anything in the typing module. For example, in `greet_people` we are using the `List` type from `typing` with our `Person` class to show the argument is a list of people.

```
from typing import List  # Import the List type

# greet_people takes a list of people as an argument
def greet_people(people: List[Person]):
    for person in people:
        greet_person(person)
```

Quiz Time! 🤓

What will this output?

```
from typing import List  # Import the List type

class Person():
    def __init__(self, name):
        self.name = name

def greet_person(person: Person):
    print(f"Hello {person.name}")

person1 = Person("Jane")
person2 = Person("Sam")
people = [person1, person2]

def greet_people(people: List[Person]):
    for person in people:
        greet_person(person)

greet_people(people)
```

1. `Hello Sam\nHello Jane`
2. `keyError: can not read name of List`
3. `Hello Jane\nHello Sam`

4. `Hello Jane\nHello Jane`

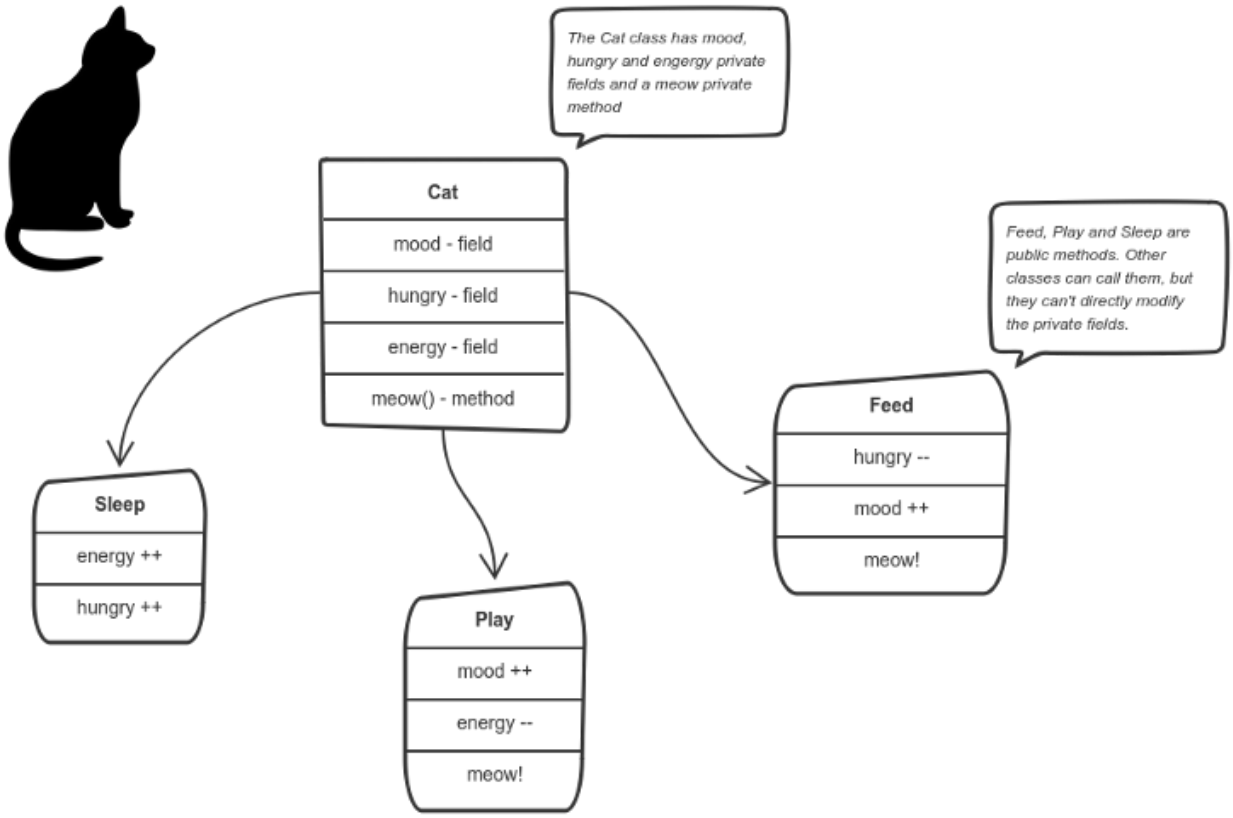
In these solutions `\n` is a newline. Answer is `3. Hello Jane\nHello Sam`

Answer: 3

The Four Principles of OOP

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Encapsulation



Encapsulation

- Grouping together of properties(props) and methods which define a *type*
  - i.e. Bundling up of data and functions that operate on that data into a consistent unit
- Controlling how properties are accessed and used
  - The props of the cat are stored inside the object itself
  - Specific methods can be used by external callers to modify or access specific values

---

Encapsulation

Encapsulation provides context to variables and functions. A dog barks while a cat meows. This can prevent you calling the wrong function with the wrong type.

encapsulation.py

```python
# Encapsulation - this is encapsulated in a class
class Cat():
    name = "Sam the cat"

    def meow(self):
        print(self.name + " says Meow!")


class Dog():
    name = "Logan the dog"

    def bark(self):
        print(self.name + " says Bark!")


c = Cat()
c.meow()

d = Dog()
d.bark()
```

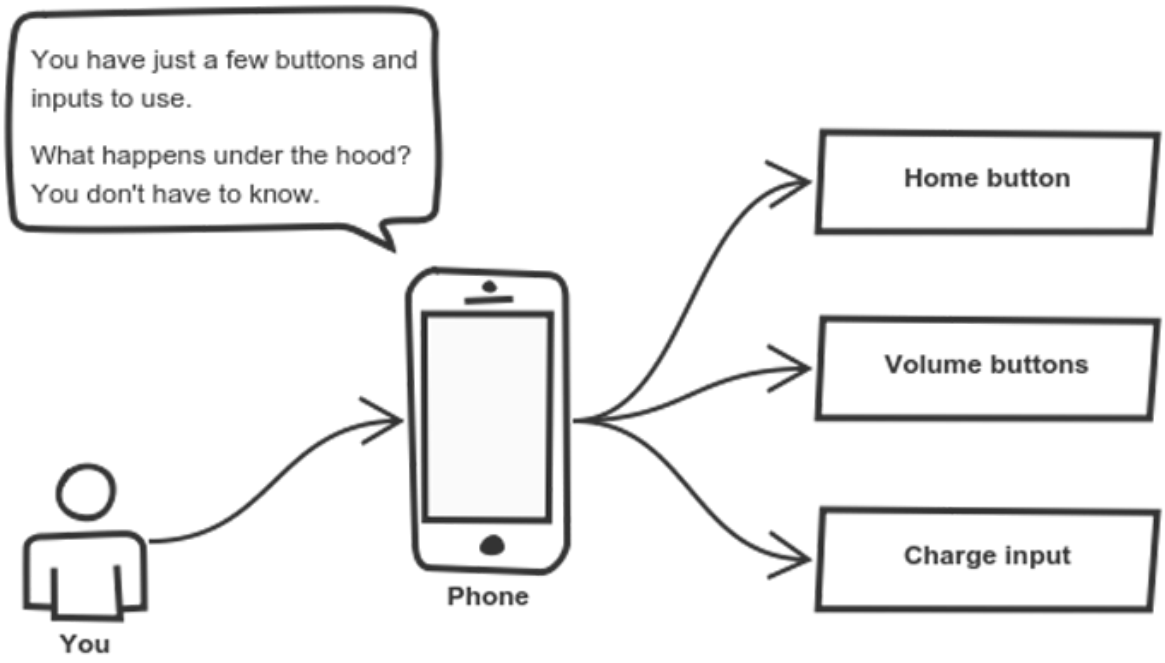
unencapsulation.py

```python
# Unencapsulated - these dictionaries are not encapsulated
cat = {
    "name": "Sam the cat",
}

dog = {
    "name": "Logan the dog",
}


def meow(cat):
    print(cat["name"] + " says Meow!")


def bark(dog):
    print(dog["name"] + " says Bark!")


meow(cat)
bark(dog)
# Theres nothing stopping us using these functions
# with the wrong dictionary.
meow(dog)
bark(cat)
```

The Four Principles of OOP

- Encapsulation ✓
- Abstraction
- Inheritance
- Polymorphism

Abstraction



Abstraction

- We do not need to know how things work in order to use them
- We just need to know that they exist, and how they should be used
- Implemenation specifics can be hidden from us and different, as long as the thing provides a consistent interface

Abstraction Example

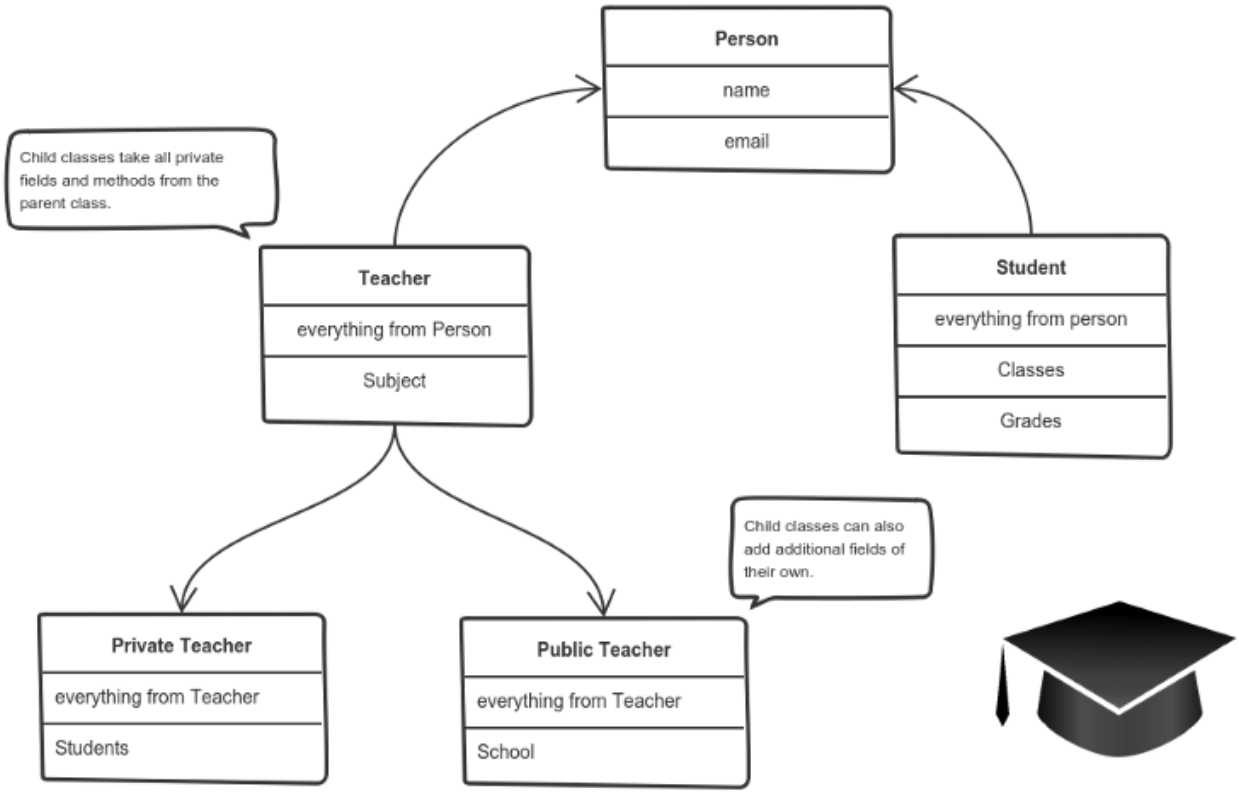
Please refer to the `abstraction.py` handout.

Read the comments: Phone is an abstract class because it contains an abstract method. Smart phone inherits from phone because its a kind of phone. The other kinds of phone also inherit from Phone. As a result all of the phones are capable of calling a friend. So the `arrange birthday party` function uses a phone, but it doesn't matter what kind of phone you use. All that matters is you can use it to call a friend.

The Four Principles of OOP

- Encapsulation ✓
- Abstraction ✓
- Inheritance
- Polymorphism

Inheritance



Inheritance

- The process by which one class takes on the attributes/methods of another
- A class inheriting from another is called a **child** class
- The class that is being inherited from is called a **parent** class
- Child classes can **override** or **extend** parent class methods to allow for different behaviours
- Child classes can add new **methods** or **properties**

This is classical inheritance as opposed to the prototypal inheritance we see in the likes of JS

Example: Define our base type

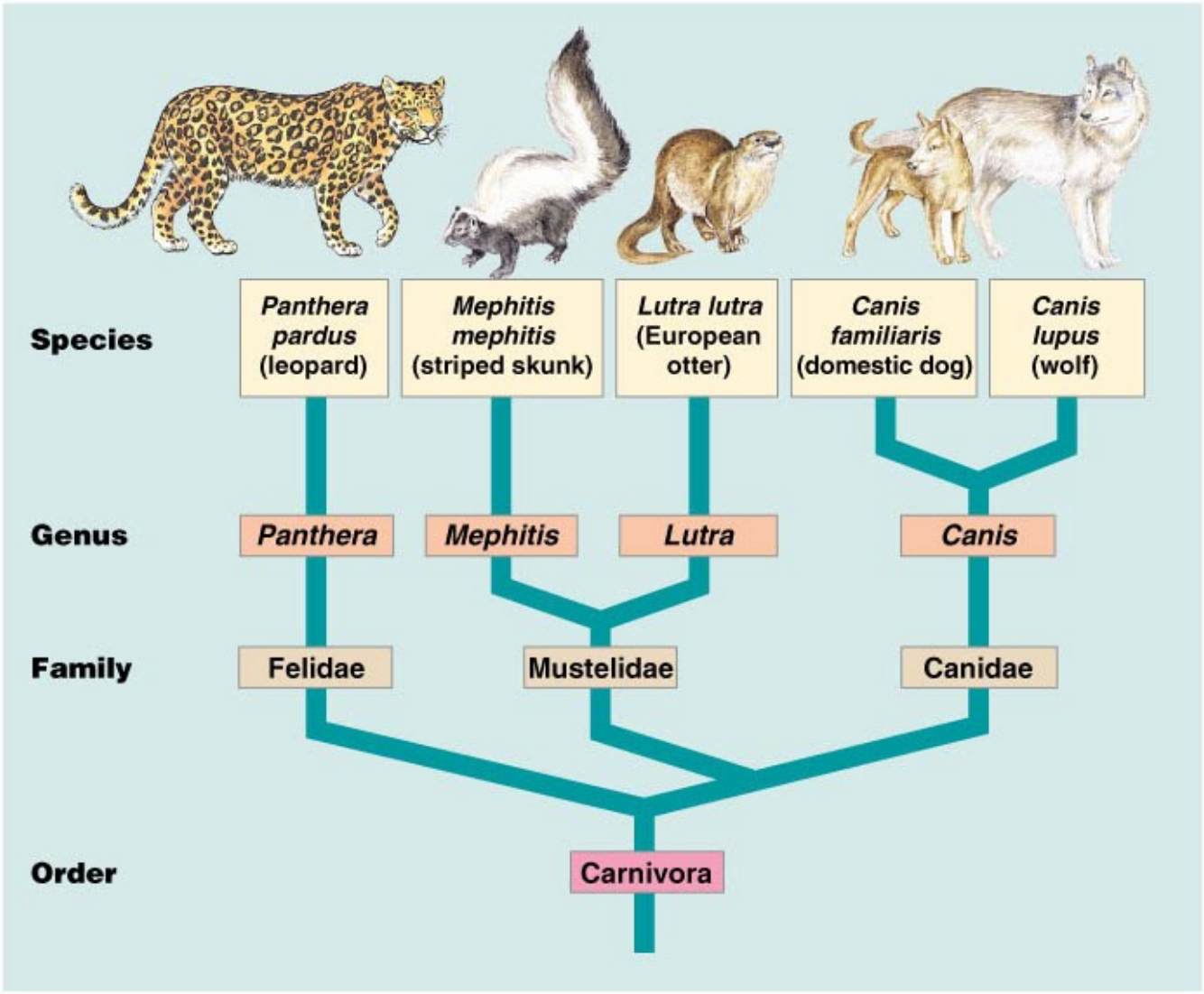
```
class Person:
    def __init__(self, name):
        self.name = name

    def describe(self):
        return f"I am {self.name}"
```

Example: Derive a sub-type

```
class Employee(Person):
    def __init__(self, name, job_title):
        self.name = name
        self.job_title = job_title

    def describe_profession(self):
        return f"I work as a {self.job_title}"
```

```
e = Employee("Rebecca", "Developer")
print(e.introduce()) # Can still introduce
print(e.describe_profession()) # But also can now describe profession
```

The Natural World



Reveal the different objects, classes / types, and the inheritance tree in this picture of taxonomies

Species -> Genus -> Family -> Order

A leopard is a type of Species, which is a type of Genus...

So when we create a Leopard, we get the entire animal kingdom attached to it and then some

What would happen if we changed some of the behaviour of the `Order` type?

It would affect everything downstream, and they would not be aware of it

This is the problem with inheritance and why if we do use it, we stick to only one level: Child -> Parent

---

Inheritance Example

Please refer to the `inheritance.py` handout.

In this example:

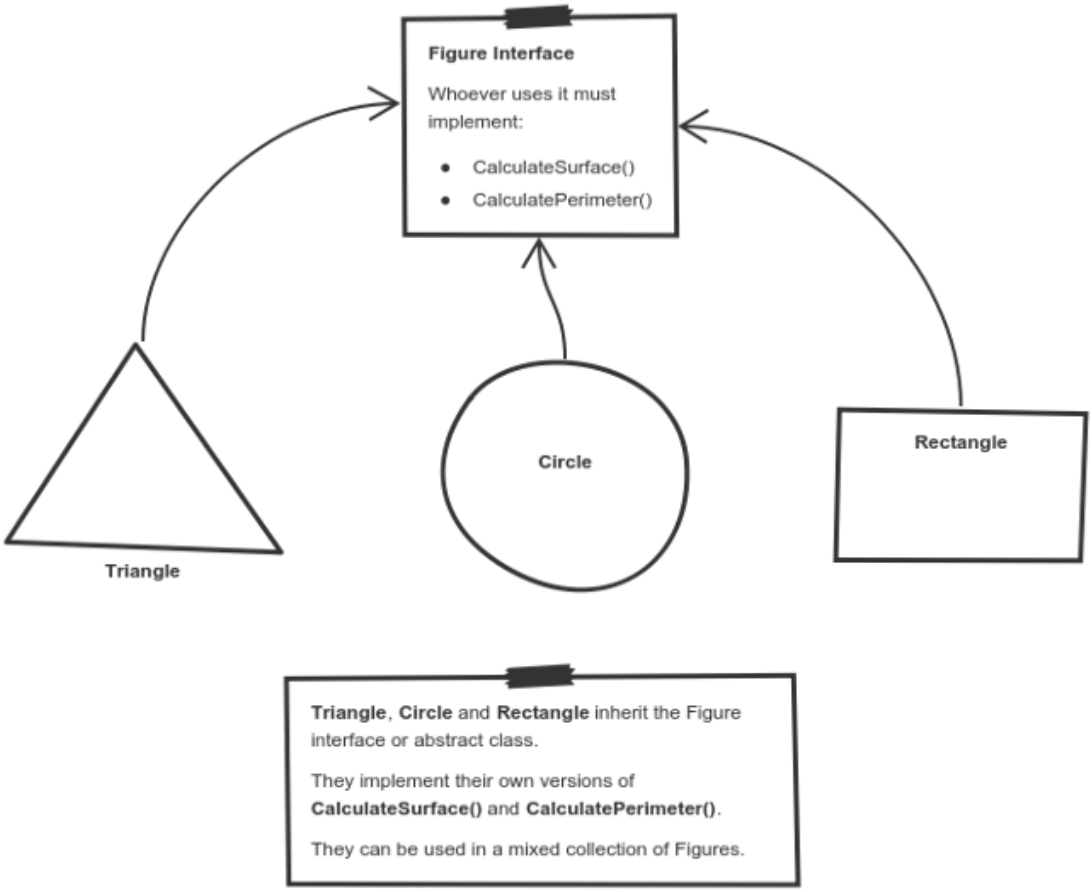
- The `Wolf` can eat because it *inherited* the `eat` function from `Animal`.
- The `eat` function eats whatever `favourite_food` is on the class.

- The `Wolf` eats `The 3 Little Piggies` whereas other Animals do not.

The Four Principles of OOP

- Encapsulation ✅
- Abstraction ✅
- Inheritance ✅
- Polymorphism

Polymorphism



Polymorphism

- Means "many forms" in Greek
- Different forms implement different behaviours for the same functionality
- Each shape class must be able to calculate it's perimeter, but the formula is different for each.

Example

Open the Python Interpreter and write this simple class:

```
>>> class MyClass:
...   pass
```

Then run this:

```
>>> c = MyClass()
>>> dir(c)
```

What do you see?

Example

`dir()` returns a list of all the members of a specified object. You didn't declare any members in `MyClass` so where did it come from?

If you run the following, you will get the same list near enough.

```
o = object()
dir(o)
```

Every class you create in Python is always a form of `object`.

This may not be the best way to reveal this, but we can think of `object` as being an abstract class, much like the figure class in the diagram.

All our types are objects, just they are different implementations of the required object methods, most of which is abstracted away from us.

Revealing that everything is a form or type of object is hopefully a good way to conclude

---

Quiz Time! 🤓

What is the output for the following piece of code?

```
class A:
    def __init__(self, x=3):
        self.x = x

class B(A):
    def __init__(self):
        super().__init__(5)
```

```
    def print(self):
        print(self.x)

obj = B()
obj.print()
```

1. 3
2. 5
3. An error is thrown

Answer: 5

Which of these best describes inheritance?

1. Controls the privacy of properties and methods of a class.
2. We do not need to know how things work in order to use them.
3. The process by which one class takes on the attributes/methods of another.
4. Different forms can implement their own versions of required methods

Answer: 3

Exercise

Instructor to distribute exercise.

Learning Objectives Revisited

- Define object oriented programming
- Identify built-in and advanced types
- Discover how we can use types with classes
- Identify the four principles of OOP
- Implement OOP using Python

Terms and Definitions Recap

Object: Where concepts are represented as objects.

Object Oriented Programming: A programming paradigm based on the concept of "objects", which can contain data and code.

Class: An extensible program-code-template for creating objects, providing initial values for state and implementations of behaviour.

Property: A special sort of class member, intermediate in functionality between a field (or data member) and a method.

Terms and Definitions Recap

Instance: A concrete occurrence of any object, existing usually during the runtime of a computer program.

Attribute: The specific value of a given instance.

Abstraction: The process of removing details or attributes to focus attention on details of greater importance.

Inheritance: The mechanism of basing an object or class upon another object.

Further Reading

- FreeCodeCamp: How to explain OOP...
- YouTube: Composition / Inheritance