# Abstract Data Types & OOP Recap

**IN608: Intermediate Application Development Concepts**

**Kaiako: Tom Clark & Grayson Orr**

# Administration

# Administration

- We will be using GitHub so make sure you have Git installed on your personal computer(s)
  - Course materials repository - https://github.com/otago-polytechnic-bit-courses/intermediate-app-dev-concepts
  - Practicals repository - https://classroom.github.com/a/2Hnb0QIq
  - Django & OpenTDB API repository - https://classroom.github.com/a/uQeihzqX
  - Django REST Framework, React & OpenTDB API repository - https://classroom.github.com/a/sSA9csHf
- Tom & Grayson will communicate with you via Microsoft Teams & Outlook

# Content

- Abstract data types
  - List
  - Tuple
  - Set
  - Dictionary
- OOP recap
  - Access modifiers
  - Encapsulation
  - Abstraction
  - Single inheritance
  - Multiple inheritance
  - Multi-level inheritance
  - Polymorphism

# Python

# Python

- Created by Guido van Rossum
- Multi-paradigm programming language
- Dynamically typed & garbage collected
- Core philosophy:
  - Beautiful is better than ugly
  - Explicit is better than implicit
  - Simple is better than complex
  - Complex is better than complicated
  - Readability counts

# Abstract Data Types

# List

- Ordered collection & mutable

```
numbers = [1, 2, 3, 4, 5] # Homogeneous
hetero = [1, 'C#', True, 2, 'Java'] # Heterogeneous
print(type(numbers)) # <class 'list'>
```

# Tuple

- Ordered collection & immutable
- It is possible to create tuples which contain mutable objects, i.e., lists

```
numbers = (1, 2, 3, 4, 5) # Homogeneous
hetero = (1, 'C#', True, 2, 'Java') # Heterogeneous
print(type(numbers)) # <class 'tuple'>
```

# Set

- Unordered collection & mutable
- Contains no duplicate elements
- When we print `hetero`, why is `True` not contained in the output?

```
numbers = {1, 2, 3, 4, 4} # Homogeneous
hetero = {1, 'C#', True, 2, 2} # Heterogeneous
print(type(numbers)) # <class 'set'>
print(numbers) # {1, 2, 3, 4}
print(hetero) # {'C#', 1, 2}
```

# Dictionary

- Unordered collection & mutable
- Key/value pairs

```
ig_user_1 = {'username': 'john_doe', 'active': False, 'followers': 150}
ig_user_2 = {'username': 'jane_doe', 'active': True, 'followers': 500}
print(type(ig_user_1)) # <class 'dict'>
print(ig_user_1['username']) # john_doe
print(ig_user_2['followers']) # 500
```

# Programming Activity (30 Minutes)

# Jupyter Notebook

- Open-source web application
- Create & share documents that contain live code, equations, visualizations & text
- Resource: https://jupyter.org/install.html

# Programming Activity

- Please open 01-practical.ipynb
- Please **ONLY** answer questions 1-5
- We will go through the solutions after 30 minutes

# Solutions

# OOP Recap

# Access Modifiers - Public

- By default, all class members are public

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

def main():
    persian = Cat('Tom', 'persian')
    persian.name = 'Jerry'
    print(f'My {persian.breed}\'s name is {persian.name}')

if __name__ == '__main__':
    main() # My persian's name is Jerry
```

# Access Modifiers - Protected

- Convention to make a class member protected - single underscore

```
class Cat:
    def __init__(self, name, breed):
        self._name = name
        self._breed = breed

def main():
    persian = Cat('Tom', 'persian')
    persian._name = 'Jerry'
    print(f'My {persian._breed}\'s name is {persian._name}')

if __name__ == '__main__':
    main() # My persian's name is Jerry
```

# Access Modifiers - Private

- Convention to make a class member protected - double underscore
- Name mangling

```
class Cat:
    def __init__(self, name, breed):
        self.__name = name
        self.__breed = breed

def main():
    persian = Cat('Tom', 'persian')
    persian._Cat__name = 'Jerry'
    print(f'My {persian._Cat__breed}\'s name is {persian._Cat__name}')

if __name__ == '__main__':
    main() # My persian's name is Jerry
```

# Encapsulation - @property Decorator

- The property object has getter, setter & deleter methods usable as decorators
- Resource: https://docs.python.org/3/library/functions.html#property

```
class Cat:
    def __init__(self, name, breed):
        self.__name = name
        self.__breed = breed

    @property
    def name(self):
        return self.__name

    @property
    def breed(self):
        return self.__breed

    @name.setter
    def name(self, name):
        self.__name = name

def main():
    persian = Cat('Tom', 'persian')
    persian.name = 'Jerry'
    print(f'My {persian.breed}\'s name is {persian.name}')

if __name__ == '__main__':
    main() # My persian's name is Jerry
```

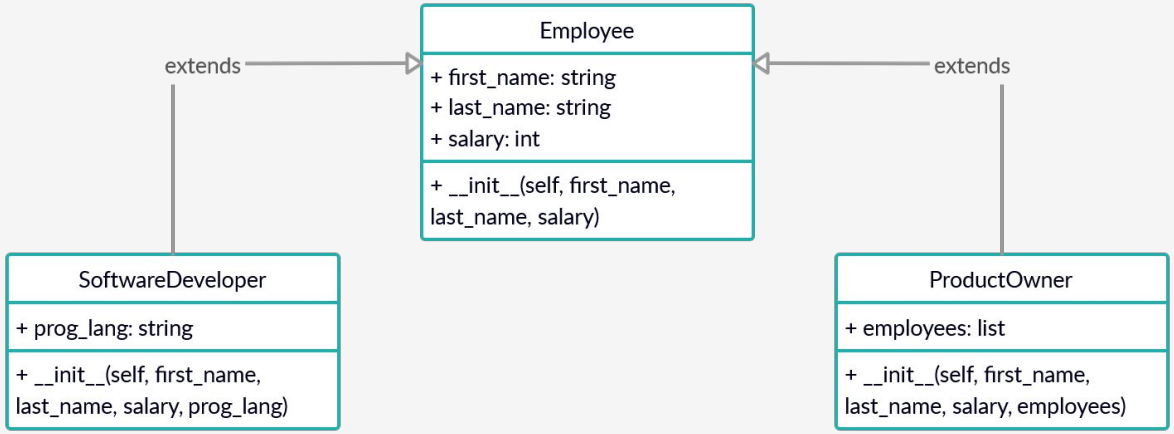
# Abstraction

- abc/Abstract Base Classes module
- Resource: https://docs.python.org/3/library/abc.html

```
from abc import ABC, abstractmethod

class Payment(ABC):
    def __init__(self, amount):
        self.amount = amount

    @abstractmethod
    def payment(self):
        raise NotImplementedError

class Eftpos(Payment):
    def __init__(self, amount):
        super().__init__(amount)

    def payment(self):
        return f'${self.amount} paid with eftpos'

class Cash(Payment):
    def __init__(self, amount):
        super().__init__(amount)

    def payment(self):
        return f'${self.amount} paid with cash'

def main():
    eftpos = Eftpos(150)
    print(eftpos.payment())
    cash = Cash(75)
    print(cash.payment())

if __name__ == '__main__':
    main() # $150 paid with eftpos
           # $75 paid with cash
```

# Single Inheritance

- Consider the following:



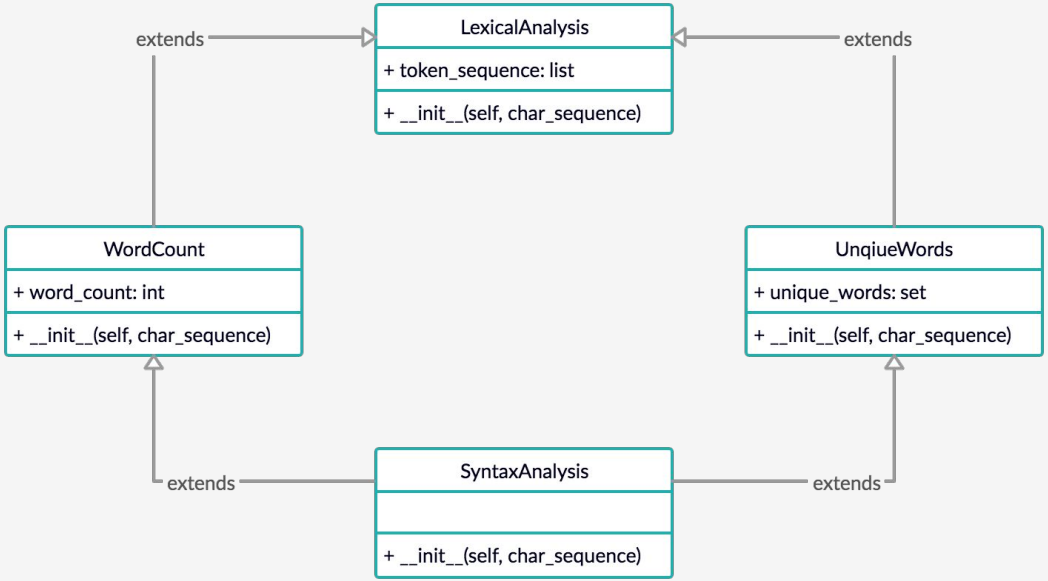
# Single Inheritance

- SoftwareDeveloper class & ProductOwner class extend Employee class
- Resource: https://docs.python.org/3/tutorial/classes.html#inheritance

```
class Employee:
    def __init__(self, first_name, last_name, salary):
        self.first_name = first_name
        self.last_name = last_name
        self.salary = salary

class SoftwareDeveloper(Employee):
    def __init__(self, first_name, last_name, salary, prog_lang):
        super().__init__(first_name, last_name, salary)
        self.prog_lang = prog_lang

class ProductOwner(Employee):
    def __init__(self, first_name, last_name, salary, employees=None):
        super().__init__(first_name, last_name, salary)
        if employees is None:
            self.employees = []
        else:
            self.employees = employees

def main():
    sft_dev_1 = SoftwareDeveloper('Alfredo', 'Boyle', 50000, 'C#')
    sft_dev_2 = SoftwareDeveloper('Malik', 'Martin', 55000, 'JavaScript')
    prdt_owr = ProductOwner('Lillian', 'Cunningham', 100000, [sft_dev_1, sft_dev_2])
    for e in prdt_owr.employees:
        print(f'{e.first_name} {e.last_name}')

if __name__ == '__main__':
    main() # Alfredo Boyle
           # Malik Martin
```

# Multiple Inheritance

- Consider the following:



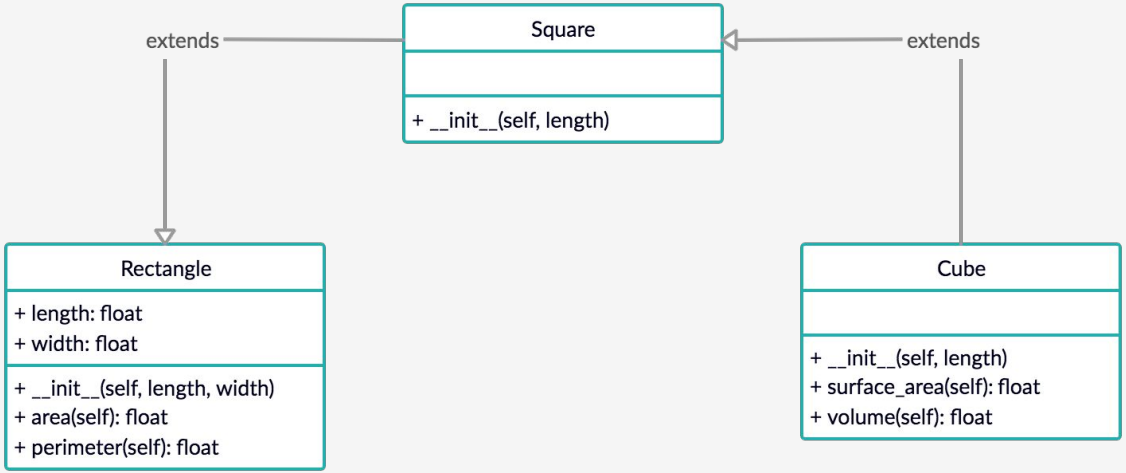
# Multiple Inheritance

- WordCount class & UnqiueWords class extend LexicalAnalysis class
- SyntaxAnalysis class extends WordCount class & UniqueWords class
- C# & Java do not support multiple inheritance
- Resource: https://docs.python.org/3/tutorial/classes.html#multiple-inheritance

```
class LexicalAnalysis:
    def __init__(self, char_sequence):
        self.token_sequence = char_sequence.split()

class WordCount(LexicalAnalysis):
    def __init__(self, char_sequence):
        super().__init__(char_sequence)
        self.word_count = len(self.token_sequence)

class UniqueWords(LexicalAnalysis):
    def __init__(self, char_sequence):
        super().__init__(char_sequence)
        self.unique_words = set(self.token_sequence)

class SyntaxAnalysis(WordCount, UniqueWords):
    def __init__(self, char_sequence):
        super().__init__(char_sequence)

def main():
    syntax_analysis = SyntaxAnalysis('I was walking down the road and I saw...a donkey, Hee Haw!')
    print(syntax_analysis.word_count)
    print(syntax_analysis.unique_words)

if __name__ == '__main__':
    main() # 12
           # {'I', 'was', 'walking', 'down', 'the', 'road,', 'and', 'saw...a', 'donkey,', 'Hee', 'Haw!'}
```

# Multi-Level Inheritance

- Consider the following:



# Multi-Level Inheritance

- Square class extends Rectangle class
- Cube class extends Square class

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * (self.length + self.width)

class Square(Rectangle):
    def __init__(self, length):
        super().__init__(length, length)

class Cube(Square):
    def __init__(self, length):
        super().__init__(length)

    def surface_area(self):
        return super().area() * 6

    def volume(self):
        return super().area() * self.length

def main():
    cube = Cube(4.5)
    print(cube.surface_area())

if __name__ == '__main__':
    main() # 121.5
```

# Polymorphism - Subtyping

- Subtype/inclusion polymorphism
- Country class has three subtypes - NewZealand class, Brazil class & Canada class
- Liskov Substitution principle - we will look at this next week
- What is the output?

```
class Country:
    def capital(self):
        raise NotImplementedError

class NewZealand(Country):
    def capital(self):
        return 'Wellington is the capital of New Zealand.'

class Brazil(Country):
    def capital(self):
        return 'Brasilia is the capital of Brazil.'

class Canada(Country):
    pass

def main():
    nzl = NewZealand()
    bra = Brazil()
    can = Canada()
    for country in (nzl, bra, can):
        print(country.capital())

if __name__ == '__main__':
    main()
```

# Polymorphism - Duck Typing

- If it walks like a 🦆 & quacks like a 🦆, then it must be a 🦆

```
class NewZealand:
    def capital(self):
        return 'Wellington is the capital of New Zealand.'

class Brazil:
    def capital(self):
        return 'Brasilia is the capital of Brazil.'

class Canada:
    pass

def main():
    nzl = NewZealand()
    bra = Brazil()
    can = Canada()
    for country in (nzl, bra, can):
        print(country.capital())

if __name__ == '__main__':
    main()
```