Python 1: 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

Today’s Content

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

Python

Overview

- 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
- Resource: https://www.python.org

Abstract Data Types

List

- Ordered collection & mutable

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

Tuple

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

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

Set

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

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

Dictionary

- Unordered collection & mutable
- Key/value pairs

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

Programming Activity
(30 Minutes)

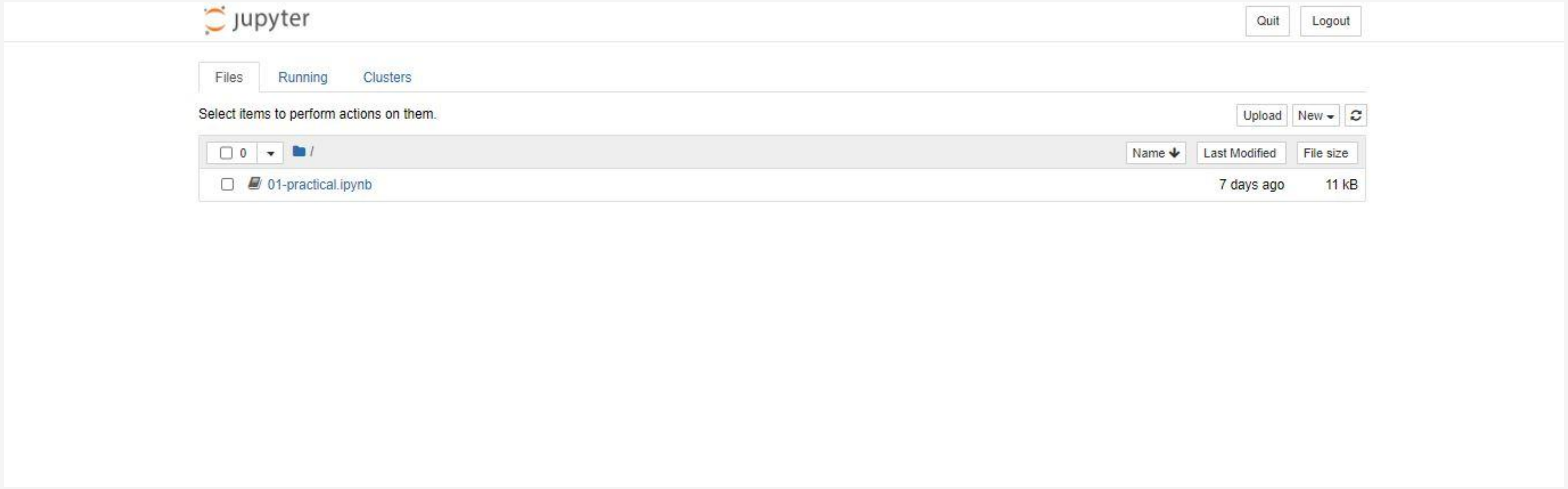
Jupyter Notebook

- Open-source web application
- Create & share documents that contain live code, equations, visualizations & text
- Requires Python
- We suggest that you create a virtual environment for your Jupyter Notebooks. There are plenty of resources online on how to set this up
- Resources:
    - https://www.python.org/downloads
    - https://jupyter.org/install.html

Programming Activity

- Click on the practicals link in slide three to generate your practicals repository
- Once generated, clone your practicals repository to your computer (student drive)
- Copy 01-practical.ipynb from the course materials repository into your practicals repository
- Open up the Anaconda Prompt (it should be install on all lab computers) & `cd` to your practicals repository
- Run the following command: `jupyter notebook`

Programming Activity



Select items to perform actions on them.

01-practical.ipynb	7 days ago	11 kB

Programming Activity

- Please open 01-practical.ipynb
- Please **ONLY** answer questions 1-4
- We will go through the solutions after 30 minutes

Solutions

Lecture Code

- All code examples from the today’s lecture - 01-lecture-code.ipynb

OOP Recap

Access Modifiers - Public

- By default, all class members are public

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

    def __str__(self):
        return f'My {self.breed}\'s name is {self.name}'

def main():
    persian = Cat('Tom', 'persian')
    persian.name = 'Jerry'
    print(persian)

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 __str__(self):
        return f'My {self._breed}\'s name is {self._name}'

def main():
    persian = Cat('Tom', 'persian')
    persian._name = 'Jerry'
    print(persian)

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

Access Modifiers - Private

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

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

    def __str__(self):
        return f'My {self.__breed}\'s name is {self.__name}'

def main():
    persian = Cat('Tom', 'persian')
    persian._Cat__name = 'Jerry'
    print(persian)

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

```python
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 __str__(self):
        return f'My {self.__breed}\'s name is {self.__name}'

def main():
    persian = Cat('Tom', 'persian')
    persian.name = 'Jerry'
    print(persian)

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

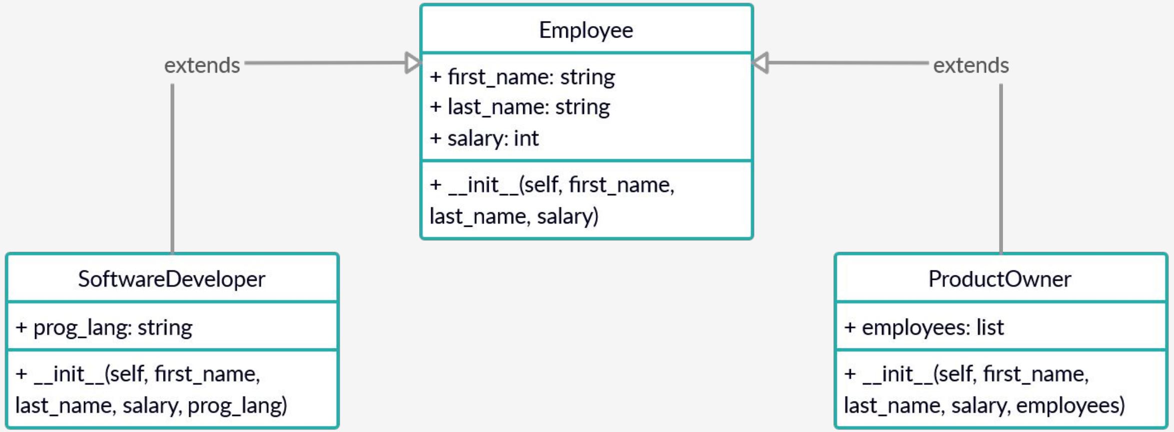
Abstraction

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

```python
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)
    cash = Cash(75)
    print(eftpos.payment())
    print(cash.payment())

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

Single Inheritance

- Consider the following:


Employee
+ first_name: string
+ last_name: string
+ salary: int
+ __init__(self, first_name, last_name, salary)
extends
extends
SoftwareDeveloper
+ prog_lang: string
+ __init__(self, first_name, last_name, salary, prog_lang)
ProductOwner
+ employees: list
+ __init__(self, first_name, last_name, salary, employees)

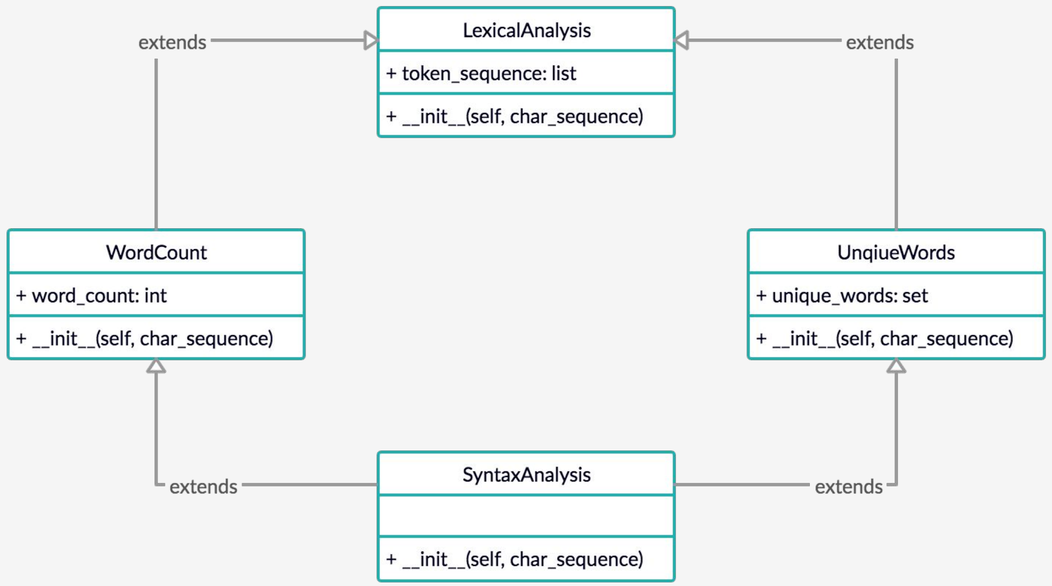
Single Inheritance

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

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

    def __str__(self):
        return f'{self.first_name} {self.last_name}'

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):
        super().__init__(first_name, last_name, salary)
        self.employees = employees

    def show_employees(self):
        for employee in self.employees:
            print(employee)

def main():
    sft_dev_one = SoftwareDeveloper('Alfredo', 'Boyle', 50000, 'C#')
    sft_dev_two = SoftwareDeveloper('Malik', 'Martin', 55000, 'JavaScript')
    prdt_owr = ProductOwner('Lillian', 'Cunningham', 100000, [sft_dev_one, sft_dev_two])
    prdt_owr.show_employees()

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

Multiple Inheritance

- Consider the following:



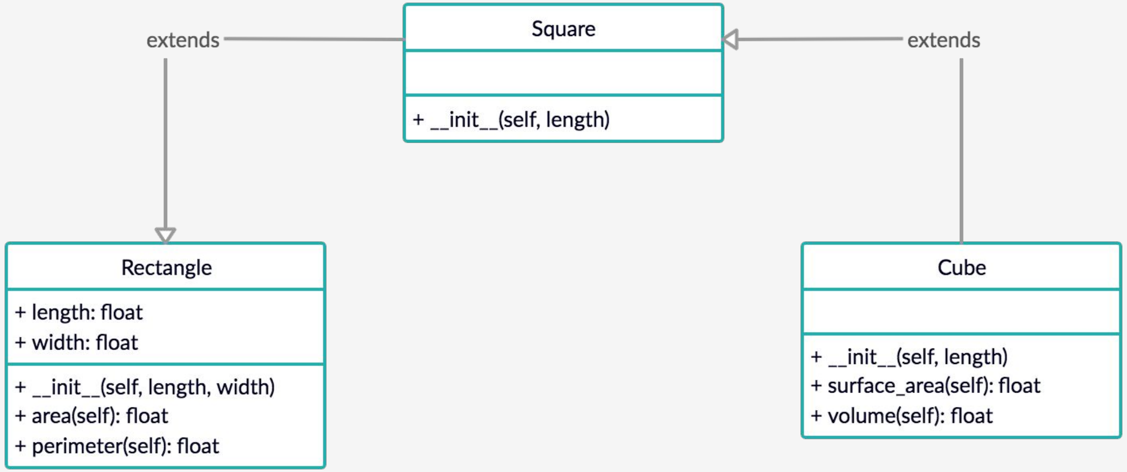
Multiple Inheritance

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

```python
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` extends `Rectangle`
- `Cube` extends `Square`

```python
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`, `Brazil` & `Canada`
- Liskov Substitution principle - we will look at this next week
- What is the output?

```python
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 🦆

```python
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()
```