Lecture 15: Basic Data Structures & Comprehensions
IN628: Programming 4
Semester One, 2020

Kaiako: Grayson Orr

Te Kura Matatini ki Otago, Ōtepoti, Aotearoa

Object-Oriented Programming Principles

The four principles of OOP:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Encapsulation

- Bundling of attributes & methods
- Used to hide the internal details of a class
- Access modifiers/specifiers

Encapsulation

- Public attributes

```python
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
```

Encapsulation

- Private attributes
- Getters & setters (Javaesque)

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

    def get_name(self):
        return self.__name

    def get_breed(self):
        return self.__breed

    def set_name(self, name):
        self.__name = name

    def set_breed(self, breed):
        self.__breed = breed

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

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

Encapsulation

- @property, @attribute.setter & @attribute.deleter (Pythonic)

```
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

    @name.deleter
    def name(self):
        print('Deleting name')
        del self.__name

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

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

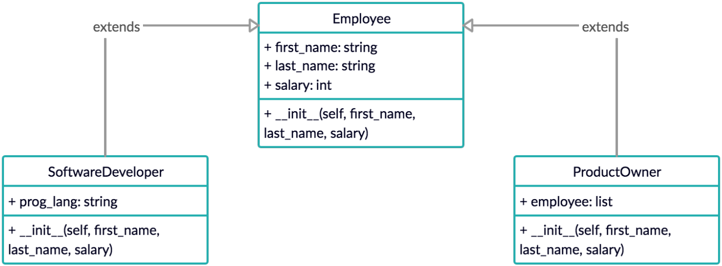
Abstraction

- abc module
- @abstractmethod

```python
from abc import ABC, abstractmethod

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

    @abstractmethod
    def payment(self):
        pass

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

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

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

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

def main():
    credit_card = CreditCard(150)
    print(credit_card.payment())
    cash = Cash(75)
    print(cash.payment())

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

Single Inheritance: UML

- Consider the following UML diagram:



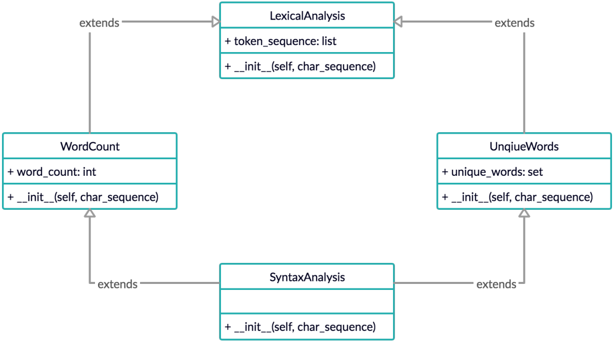
Single Inheritance

- SoftwareDeveloper & ProductOwner inherits from Employee

```python
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: UML

- Consider the following UML diagram:



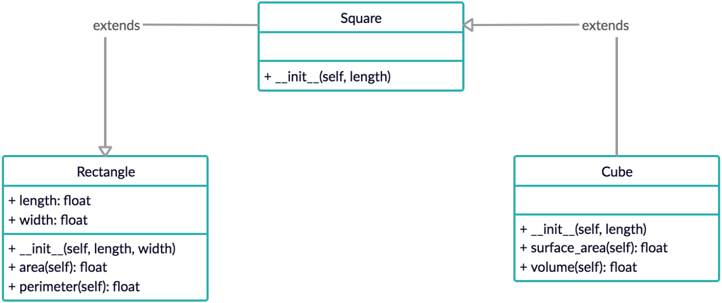
Multiple Inheritance

- WordCount & UniqueWords inherits from LexicalAnalysis
- SyntaxAnalysis inherits from WordCount & UniqueWords

```
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(
      'Peter Piper picked a peck of pickled peppers; A peck of pickled peppers Peter Piper picked')
    print(syntax_analysis.word_count)
    print(syntax_analysis.unique_words)

if __name__ == '__main__':
    main()  # 16
            # {'peppers;', 'peppers', 'a', 'picked', 'Piper', 'pickled', 'of', 'peck', 'Peter', 'A'}
```

Multi-Level Inheritance: UML

- Consider the following UML diagram:



Multi-Level Inheritance

- Square inherits from Rectangle
- Cube inherits from 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

- Poly (many)
- Morphism (forms)
- A single interface to entities of different types
- Subtyping
- Duck typing

Polymorphism

- Subtyping
- Liskov substitution principle
- NotImplementedError exception

```python
class Country:
    def capital(self):
        raise NotImplementedError('capital was not implemented.')

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()
    for country in (nzl, bra):
        print(country.capital())

if __name__ == '__main__':
    main()  # Wellington is the capital of New Zealand.
            # Brasilia is the capital of Brazil.
            # NotImplementedError: capital was not implemented.
```

Polymorphism

- Duck typing

```
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()  # Wellington is the capital of New Zealand.
            # Brasilia is the capital of Brazil.
            # AttributeError: 'Canada' object has no attribute 'capital'
```

Basic Data Structures

- List
- Tuple
- Set
- Dictionary
- Linked List
- Stack
- Queue

List

- Mutable
- Ordered sequence of elements

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

List

- Operations:
  - append
  - clear
  - copy
  - count
  - extend
  - index
  - insert
  - pop
  - remove
  - reverse
  - sort

Tuple

- Immutable
- Ordered sequence of elements

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

Tuple

- Operations:
  - count
  - index

Set

- Immutable
- Unordered sequence of unique elements

```
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)  # {1, 'C#', 2}
```

Set

- Operations:
  - add
  - clear
  - copy
  - difference
  - difference_update
  - discard
  - intersection
  - intersection_update
  - isdisjoint
  - issubset
  - issuperset
  - pop
  - remove
  - symmetric_difference
  - symmetric_difference_update
  - union
  - update

Dictionary

- Mutable
- Unordered sequence of 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
```

Dictionary

- Operations:
  - clear
  - copy
  - fromkeys
  - get
  - items
  - keys
  - pop
  - popitem
  - setdefault
  - update
  - values

Slicing

- Positive sequence slicing

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
start_slice_numbers = numbers[2:]
end_slice_numbers = numbers[2:6]
step_slice_numbers = numbers[2::2]
print(start_slice_numbers)  # [3, 4, 5, 6, 7, 8, 9, 10]
print(end_slice_numbers)  # [3, 4, 5, 6]
print(step_slice_numbers)  # [3, 5, 7, 9]
```

0	1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9	10

Slicing

- Negative sequence slicing

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
neg_start_slice_numbers = numbers[-2:]
neg_end_slice_numbers = numbers[2:-6]
neg_step_slice_numbers = numbers[2::-2]
print(neg_start_slice_numbers)  # [9, 10]
print(neg_end_slice_numbers)  # [3, 4]
print(neg_step_slice_numbers)  # [3, 1]
```

-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
1	2	3	4	5	6	7	8	9	10

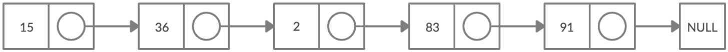
Slicing

- Computation/running time

```
from timeit import timeit

def for_loop_sentence(sentence):
    reverse_sentence = ''
    for s in sentence:
        reverse_sentence = s + reverse_sentence
    return reverse_sentence

def recursion_sentence(sentence):
    if len(sentence) == 0:
        return sentence
    else:
        return recursion_sentence(sentence[1:]) + sentence[0]

def slice_sentence(sentence):
    return sentence[::-1]

print(timeit('for_loop_sentence("Peter Piper picked a peck of pickled peppers")',
             setup='from __main__ import for_loop_sentence', number=1_000_000))  # 4.176007382999842
print(timeit('recursion_sentence("Peter Piper picked a peck of pickled peppers")',
             setup='from __main__ import recursion_sentence', number=1_000_000))  # 19.085508474000108
print(timeit('slice_sentence("Peter Piper picked a peck of pickled peppers")',
             setup='from __main__ import slice_sentence', number=1_000_000))  # 0.31656659000009313
```

Linked List

- Elements are stored at non-contiguous memory locations
- Each node contains data & a reference to the next node
- Efficient insertion & deletion of elements
- Arrays have better cache locality



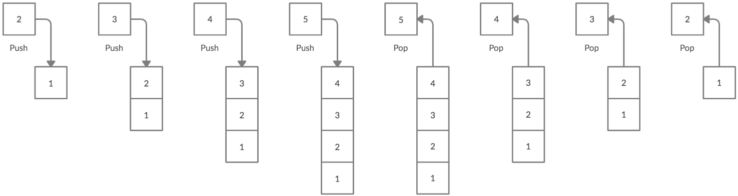
Linked List

- Implementations:
  - Singly
  - Doubly
  - Multiply
  - Circular
- Time complexity

Algorithm	**Average**	**Worst Case**
Access	O(n)	O(n)
Search	O(n)	O(n)
Insert	O(1)	O(1)
Delete	O(1)	O(1)

Stack

- LIFO (last in, first out)
- Operations:
  - is_empty
  - is_full
  - push
  - pop
  - peek



Stack

- Implementations:
  - Array
  - Linked list (singly)
- Time complexity

Algorithm	**Average**	**Worst Case**
Access	O(n)	O(n)
Search	O(n)	O(n)
Insert	O(1)	O(1)
Delete	O(1)	O(1)

Stack

```python
class Stack:
    def __init__(self):
        self.stack = []

    def is_empty(self):
        pass

    def push(self, item):
        pass

    def pop(self):
        pass

    def peek(self):
        pass

def main():
    stack = Stack()

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

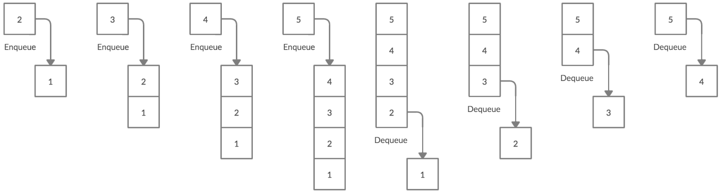
Stack

- Balanced parentheses problem

```
def balanced_parentheses(string):
    stack = []
    opening_parentheses = ['[', '(', '{']
    closing_parentheses = [']', ')', '}']

    for s in string:
        if s in opening_parentheses:
            stack.append(s)
        elif s in closing_parentheses:
            idx = closing_parentheses.index(s)
            if len(stack) > 0 and opening_parentheses[idx] == stack[len(stack) - 1]:
                stack.pop()
            else:
                return False
    if len(stack) == 0:
        return True

def main():
    print(balanced_parentheses('[[[({})]]][]'))
    print(balanced_parentheses('[[[({})]]][}'))

if __name__ == '__main__':
    main()  # True
            # False
```

Queue

- FIFO (first in, first out)
- Operations:
  - is_empty
  - is_full
  - enqueue
  - dequeue
  - size



Queue

- Implementations:
  - Double-ended queue (deque)
  - Linked list (singly & doubly)
- Time complexity

Algorithm	**Average**	**Worst Case**
Access	O(n)	O(n)
Search	O(n)	O(n)
Insert	O(1)	O(1)
Delete	O(1)	O(1)

Queue

```python
class Queue:
    def __init__(self):
        self.queue = []

    def is_empty(self):
        pass

    def enqueue(self, item):
        pass

    def dequeue(self):
        pass

    def size(self):
        pass

def main():
    queue = Queue()

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

Queue

- Balanced parentheses problem

```
def balanced_parentheses(string):
    queue = []
    opening_parentheses = tuple('[({')
    closing_parentheses = tuple('])}')
    map_parentheses = dict(zip(opening_parentheses, closing_parentheses))

    for s in string:
        if s in opening_parentheses:
            queue.append(map_parentheses[s])
        elif s in closing_parentheses:
            if not queue or s != queue.pop():
                return False
    return True

def main():
    print(balanced_parentheses('[[[({})]]][]'))
    print(balanced_parentheses('[[[({})]]][}'))

if __name__ == '__main__':
    main()  # True
            # False
```

Comprehensions

- Creates a sequence based on existing collections
- Follows the form of the mathematical set-builder notation
- Types of comprehensions:
  - List
  - Set
  - Dictionary

Set-Builder Notation

- Consider the following set-builder notation:

$$S = \{2 \cdot x \mid x \in \mathbb{N},\ x^2 > 3\}$$

- Output expression - $2 \cdot x$
- Variable - x
- Input set - $\mathbb{N}$
- Predicate - $x^2 > 3$

List Comprehension

- Consider the following code:

```
string = '123 Hi 456'
numbers = []
for s in string:
    if s.isdigit():
        numbers.append(int(s))
print(numbers)  # [1, 2, 3, 4, 5, 6]
```

List Comprehension

- Solution:

```python
string = '123 Hi 456'
numbers = [int(s) for s in string if s.isdigit()]
print(numbers)  # [1, 2, 3, 4, 5, 6]
```

Set Comprehension

- Consider the following code:

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

def main():
    cats = [
        Cat('Persian', True),
        Cat('Persian', True),
        Cat('Maine Coon', False),
        Cat('Siamese', False),
        Cat('Turkish Angora', True),
        Cat('Briman', False)
    ]

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

Set Comprehension

- Solution:

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

def main():
    cats = [
        Cat('Birman', True),
        Cat('Birman', True),
        Cat('Maine Coon', False),
        Cat('Persian', False),
        Cat('Ragdoll', False),
        Cat('Siamese', True)
    ]
    active_cats = {c.breed for c in cats if c.is_active}
    print(active_cats)

if __name__ == '__main__':
    main()  # {'Birman', 'Siamese'}
```

Dictionary Comprehension

- Consider the following code:

```
fruit_cost = {'apple': 0.89, 'banana': 0.75, 'orange': 0.60, 'pineapple': 3.50}
double_fruit_cost = {}
for (k, v) in fruit_cost.items():
    double_fruit_cost[k] = v * 2
print(double_fruit_cost)  # {'apple': 1.78, 'banana': 1.5, 'orange': 1.2, 'pineapple': 7.0}
```

Dictionary Comprehension

- Solution:

```
fruit_cost = {'apple': 0.89, 'banana': 0.75, 'orange': 0.60, 'pineapple': 3.50}
double_fruit_cost = {k: v * 2 for (k, v) in fruit_cost.items()}
print(double_fruit_cost)  # {'apple': 1.78, 'banana': 1.5, 'orange': 1.2, 'pineapple': 7.0}
```

Jupyter Notebook

- Open-source web application
- Create & share documents containing live code
- Click here to view the **Upload Jupyter Notebook** video

PEP 8

- Style guide for Python code
- Code is read much more often than it is written
- Python's core philosophy:
  - Beautiful is better than ugly
  - Explicit is better than implicit
  - Simple is better than complex
  - Complex is better than complicated
  - Readability counts
- Click here to view the **PEP 8 Style Guide**