

6.1 - Week 6 - Applied Theory

Objectives of this Applied Session

The objectives of this tutorial are to ensure we understand:

- Abstract Data Types (ADTs)
- Sets
- The SetADT

and we will also do a refresher on OOP concepts and namespaces

OOPs, you're doing it again!

So before we begin, we should test our knowledge on how familiar we are with Object Oriented Programming concepts.

Lets go ahead and answer the following questions.

Question 1 *Submitted Aug 30th 2022 at 8:15:33 am*

What do you think BEST describes OOP?

- ☒ Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic
- ☐ It was written and produced by Max Martin and Rami Yacoub. The lyrics refer to a woman who views love as a game, and she decides to use that to her advantage by playing with the emotions of a boy who likes her
- ☐ It is used to show recognition of a mistake or minor accident, often as part of an apology.
- ☐ It means using functions to the best effect for creating clean and maintainable software

Question 2 *Submitted Aug 30th 2022 at 8:21:40 am*

Write a Python program to create an Abstract Base Class Vehicle with `max_speed` and `mileage` instance attributes (no arguments for the constructor)

Please use 4 spaces to indent your code.

```
from abc import ABC

class Vehicle(ABC):
    def __init__(self):
        self.max_speed = None
        self.mileage = None
```

Question 3 *Submitted Aug 30th 2022 at 8:30:54 am*

Given

```
from abc import ABC

class Vehicle(ABC):

    def __init__(self, name, max_speed, mileage):
        self.name = name
        self.max_speed = max_speed
        self.mileage = mileage
```

Create a child class Bus that will inherit all of the variables and methods of the Vehicle class.

```
class Bus(Vehicle):
    def __init__(self):
        Vehicle.__init__(self, name, max_speed, mileage)
```

Question 4 *Submitted Aug 30th 2022 at 8:36:35 am*

Now that you have your Bus class, define a magic method inside the Bus class that when called with the following code:

```
b = Bus('Volvo', 250, 15)
print(b)
```

produce the following output:

```
This is a Volvo bus. Its top speed is 250 and it has a mileage of 15.
```

```
def __str__(self):
    return f"This is a {self.name} bus. Its top speed is {self.max_speed} and it has a mileage of {self.mileage}."
```

Question 5 *Submitted Aug 30th 2022 at 8:40:01 am*

Now, we want to add a method that returns the seating capacity of the vehicle. But, the seating capacity doesn't get defined in the base class.

So, given the following:

```
from abc import abstractmethod, ABC

class Vehicle(ABC):

    def __init__(self, name, max_speed, mileage):
        self.name = name
        self.max_speed = max_speed
        self.mileage = mileage
```

```
class Bus(Vehicle):
    def __init__(self, name, max_speed, mileage):
        Vehicle.__init__(self, name, max_speed, mileage)

    def get_capacity(self):
        return 50
```

Edit the Vehicle class such that this follows the Inheritance property of OOP.

```
class Vehicle(ABC):

    def __init__(self, name, max_speed, mileage):
        self.name = name
        self.max_speed = max_speed
        self.mileage = mileage

    @abstractmethod
    def get_capacity(self):
        pass
```

Question 6 *Submitted Aug 30th 2022 at 8:42:40 am*

Alright! Nearly there. So there was one thing that was one thing that was really wrong in the code above. Let's try and spot it and how we would fix it. So, given:

```
from abc import abstractmethod, ABC

class Vehicle(ABC):

    def __init__(self, name, max_speed, mileage):
        self.name = name
        self.max_speed = max_speed
        self.mileage = mileage

    @abstractmethod
    def get_capacity(self):
        pass

class Bus(Vehicle):
    def __init__(self):
        Vehicle.__init__(self, name, max_speed, mileage)

    def get_capacity(self):
        return 50
```

How would you fix this and make it a better program?

```
from abc import abstractmethod, ABC
```

```
class Vehicle(ABC):

    def __init__(self, name, max_speed, mileage):
        self.name = name
        self.max_speed = max_speed
        self.mileage = mileage

    @abstractmethod
    def get_capacity(self):
        pass

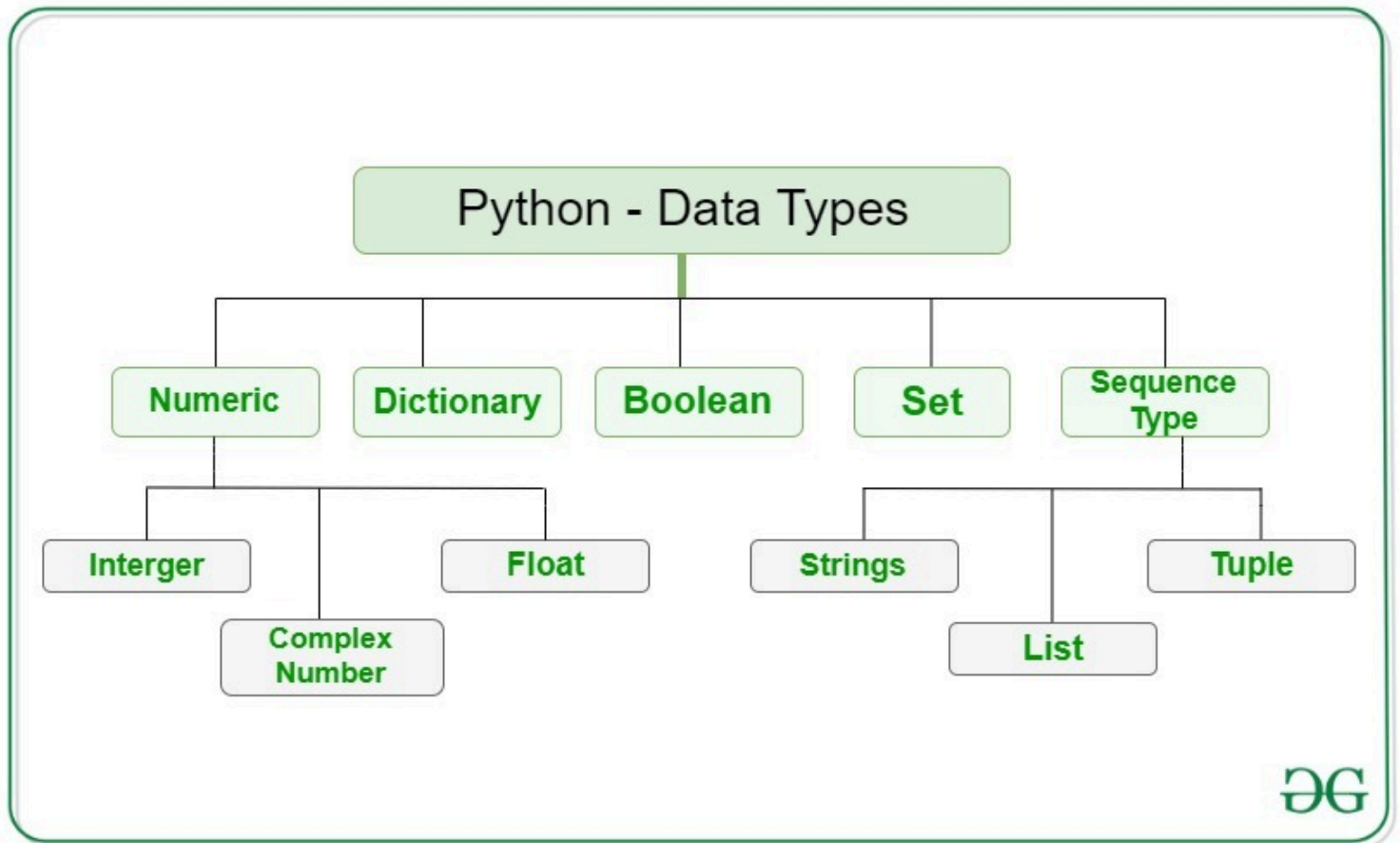
class Bus(Vehicle):
    CAPACITY = 50
    def __init__(self):
        Vehicle.__init__(self, name, max_speed, mileage)

    def get_capacity(self):
        return Bus.CAPACITY
```

An Intro to Abstract Data Types

Before we can jump into what an Abstract Data Type is, let's revise data types.

What is a data type?



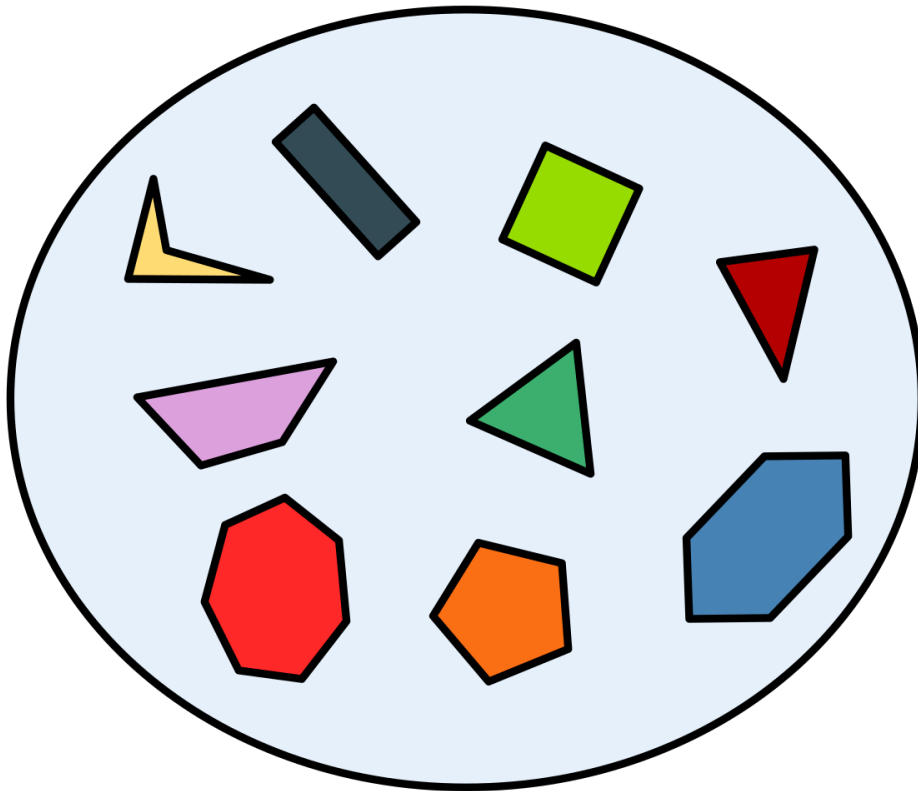
Data types such as int, float, double, long, etc. are considered to be in-built data types and we can perform basic operations with them such as addition, subtraction, division, multiplication, etc. Now there might be a situation when we need operations for our user-defined data type which have to be defined. These operations can be defined only as and when we require them. So, in order to simplify the process of solving problems, we can create data structures along with their operations, and such data structures that are not in-built are known as Abstract Data Type (ADT).

What is an ADT

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of values and a set of operations. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation-independent view.

The process of providing only the essentials and hiding the details is known as **abstraction**.

The SetADT



Sets are a type of abstract data type that allows you to store a list of non-repeated values. Their name derives from the mathematical concept of finite sets.

Unlike an array, sets are unordered and unindexed. You can think about sets as a room full of people you know. They can move around the room, changing order, without altering the set of people in that room. Plus, there are no duplicate people (unless you know someone who has cloned themselves). These are the two properties of a set: the data is unordered and it is not duplicated.

Sets have the most impact on mathematical set theory. These theories are used in many kinds of proofs, structures, and abstract algebra. Creating relations from different sets and codomains is also an important application of sets.

In computer science, set theory is useful if you need to collect data and do not care about their multiplicity or their order. Hash tables and sets are very related. In databases, especially relational databases, sets are very useful. There are many commands that find unions, intersections, and differences between different tables and sets of data.

SetADT Quiz

Lets answer the following questions to make sure our understanding of Sets is up to date.

Question 1

Sets are:

- ☐ Unordered
- ☐ Ordered
- ☐ Unique
- ☐ Non-unique

Question 2

What will be the result of the following block of code?

```
a = set()
a.add(1)
a.add(2)
a.add(1)
print(a)
```

No response

Question 3

What will be the output of the following program

```
a = set()
a.add(1)
a.add(2)

b = set()
b.add(2)
b.add(1)

print(a==b)
```


☐ True

☐ False

Question 4

Consider the following piece of code:

```
a = {1,2,3,4}
b = {2,3,5,7}

print(a & b, end=" , ")
print(a | b)
```

What will be the output?

☐ {1, 2, 3, 4} , {1, 2, 3, 4, 5, 7}

☐ {2, 3, 5, 7} , {1, 2, 3, 4, 5, 7}

☐ {2, 3} , {1, 2, 3, 4}


☐ {2, 3} , {1, 2, 3, 4, 5, 7}

Application of the SetADT

Let us implement an instance of a SetADT using arrays.

What we need to do is implement the `__and__` and the `__or__` methods for the set. You should use the given implementation of the `ASet` and edit it to define the methods.

We will then use the `ASet` class that you created to find the common distinct values from two giant list of randomly generated numbers. This will be done in the `application.py` file. You have been given two files with an enormous list of numbers(`giant_list_1.txt`, `giant_list_2.txt`). Use the `ASet` class that you built to find unique common values between the two files.

 Since the lists are really large, this might take a while to run so please be patient :) Also, this should tell you about how horrible the quadratic complexity is when it comes to large lists.

Expected Output:

```
{'17741203', '76576005'}
```

Hooray!



Alright! So You've learnt a lot about Sets!

We will now learn (and recap) about Namespaces, OO concepts and Mutability, so proceed to the rest of the exercises, but bear in mind that you might not have enough time during the applied session so continue these in your own time if you run out of time.

Exercise 1*

Examine the following code:

```
def silly() -> None:
    x = 'ping'
    def g() -> None:
        print(x)
    x = 'pong'
    def f(x) -> None:
        print(x)
    g()
    f(x)

silly()
```

We expect the **silly()** function to print "ping" and then "pong". In `Ex1.txt`, state what it will print and why. Give your explanation and reference:

- How many namespaces there are?
- What names are in those namespaces?
- How does the value that name `x` is assigned change through the execution of the code?

Exercise 2*

Consider the following situation, where a student named Juan is asked to make a Member class for a community pool. The only thing the community cares about is the pool's name, and the members' name, age and gender. So Juan writes:

```
class PoolMember:
    poolname: str = "FIT students community pool"
    name: str = ""
    age: int = 0
    gender: str = None

    def __init__(self, name: str, age: int, gender: str) -> None:
        PoolMember.name = name
        PoolMember.age = age
        PoolMember.gender = gender
```

Juan makes himself the first member of the pool, and also its admin. It all seems to go so well!

```
>>> admin = PoolMember('Juan', 18, 'male')
>>> admin.name
'Juan'
>>> admin.age
18
>>> admin.gender
'male'
>>> admin.poolname
'FIT_students_community_pool'
```

But an order comes from above. The pool has to be open to any Monash student and staff, not just FIT students. And the name needs to change before other students come! So Juan is told to change the pool name before new membership cards are made for a well-respected professor, Emilia, who will be Juan's supervisor:

```
>>> admin.poolname = "MONASH_all-inclusive_community_pool"
>>> admin.poolname
'MONASH_all-inclusive_community_pool'                                # it looks alright
>>> supervisor = PoolMember('Emilia', 26, 'female') # let's make Emilia's user
```

Phew! Disaster averted! All seems alright again, until Emilia wants to check that the pool's name was

really changed:

```
>>> supervisor.poolname
'FIT_students_community_pool'    # what, this can't be!
>>> PoolMember.poolname         # let's check the global name
'FIT_students_community_pool'    # it's the same!
```

It's still the old name! Emilia wants to find who is responsible for the mistake, so she looks at the details for the admin:

```
>>> admin.name
'Emilia'
>>> admin.age
26
>> admin.gender
'female'
```

What? Is Juan framing Emilia for his mistakes? Or is it just an unfortunate series of bugs?

In `Ex2.txt`, give an explanation for what Juan did wrong, and how to fix it. In `Ex2.py`, correct the definition of the **PoolMember** class.

Exercise 3*

Consider the following code:

```
class C1:
    m = 1
    l = 1
    def __init__(self) -> None:
        self.i = 1
        self.j = 1
        self.k = 1

class C2(C1):
    def __init__(self) -> None:
        C1.__init__(self)
        self.i = 2
        self.j = 2
        self.l = 2

class C3(C2):
    def __init__(self) -> None:
        self.i = 3
        C2.__init__(self)
        self.j = 3
        self.m = 3

a = C1()
b = C2()
c = C3()

print(a.i, b.i, c.i)
print(a.j, b.j, c.j)
print(a.k, b.k, c.k)
print(a.l, b.l, c.l)
print(a.m, b.m, c.m)
```

What will the above code print and, most importantly, why? In `Ex3.txt`, fill in the table of values, and explain each value.

Your explanation should indicate how Python dynamically decides what object needs to be printed, based on inheritance and scoping rules for each name in the print statements.

Exercise 4*

Consider the following code:

```
class Person:
    def __init__(self , name: str, surname: str, passport: str) -> None:
        self.name = name
        self.surname = surname
        self.passport = passport
    def __str__(self) -> str:
        return self.name + ", " + self.surname + ", " + str(self.passport)

class Worker(Person):
    def __init__(self , name: str, surname: str, passport: str, job: str) -> None:
        Person.__init__(self , name , surname , passport)
        self.job = job
    def __str__(self) -> str:
        return Person.__str__(self) + ", " + str(self.job)

class Athlete(Person):
    def __init__(self , name: str, surname: str, passport: str, sport: str) -> None:
        Person.__init__(self , name , surname , passport)
        self.sport = sport
    def __str__(self) -> str:
        return Person.__str__(self) + ", " + str(self.sport)
```

In `Ex4.py` define a new class **AthleteWorker** that uses inheritance to combine the attributes of a worker and an athlete and defines appropriate `__init__` and `__str__` methods. Additionally, extend the `Person` class with an `__eq__` method that considers two people equivalent if they have the same passport.

In `Ex4.txt` discuss how could this be used in your **AthleteWorker** class to determine whether two such objects are equivalent.

Exercise 5

Draw in Paint (alternatively [this](#)) a memory diagram representing the state of memory after the following instructions have been executed.

```
numbers = [1, 2, 3]
other = numbers
numbers.append(4)
x = numbers[0]
x += 1
print(other)
print(numbers)
```

In `Ex5.txt`, explain what will be printed in the end, making reference to the memory diagram.

Exercise 6

Consider the following code:

```
class Phone:
    year: int = 2012
    memory: int = 3
    network: str = '2G'

    def __init__(self, model: str, colour: str) -> None:
        self.model = model
        self.colour = colour

    def set_memory(self, memory: int) -> None:
        self.memory = memory

    def set_network(self, network: str) -> None:
        Phone.network = network

phone1 = Phone('Pony', 'Magenta')
phone2 = Phone('Tomato', 'Green')

phone2.set_network('4G')
phone2.network = '3G'

phone1.set_memory(8)
phone1.year = 2018

print(phone1.colour)
print(phone1.memory)
print(phone1.network)

print(phone2.year)
print(phone2.memory)
print(phone2.network)

print(Phone.year)
print(Phone.model)
```

What will each print statement show and, most importantly, why? In `Ex6.txt`, state what it will print and why. Give in your explanation and reference:

- How many namespaces there are?
- What names are in those namespaces?
- How do the objects assigned to names `phone1` and `phone2` change through execution?